

# SQL

3 BOOKS IN 1

The Ultimate Beginner, Intermediate & Expert Guides To  
Master SQL Programming Quickly with Practical Exercises



MARK REED

SQL - BEGINNER

1

SQL - INTERMEDIATE

2

SQL - EXPERT

3

# SQL

**3 books 1 - The Ultimate Beginner, Intermediate &  
Expert Guides To Master SQL Programming  
Quickly with Practical Exercises**

# **SQL**

**The Ultimate Beginner's Guide to Learn SQL  
Programming step by step**

**Mark Reed**

**© Copyright 2022 - All rights reserved.**

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book. Either directly or indirectly.

**Legal Notice:**

This book is copyright protected. This book is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

# Table of Contents

[Introduction](#)

[Chapter 1:](#)

[Understanding Databases](#)

[What's a Database?](#)

[Database Management Systems](#)

[Flat Files](#)

[Database Types](#)

[The Relational Database](#)

[The Object-Relational Database](#)

[Chapter 2:](#)

[Using Queries to Obtain Data](#)

[The Basics](#)

[Query Structure and the SELECT Statement](#)

[The WHERE Clause](#)

[Using ORDER BY](#)

[Subqueries](#)

[Filtering Data with Subqueries](#)

[Derived Tables with Subqueries](#)

[Table Expressions](#)

[Cross Tabulations](#)

[Tabulating](#)

[Chapter 3:](#)

[The Data Definition Language \(DDL\)](#)

[Using DDL to Create](#)

[Adding Foreign Keys with ALTER](#)

[Creating Foreign Key DDL](#)

[Unique Constraints](#)

[Deleting Tables and Databases](#)

[How to Create Views](#)

[Chapter 4:](#)

[SQL Joins and Union](#)

[INNER JOIN](#)

[RIGHT JOIN](#)

[LEFT JOIN](#)

[The UNION Statement](#)

[The UNION ALL Statement](#)

[Chapter 5:](#)

[Data Integrity](#)

[Integrity Constraints](#)

[The Not Null Constraint](#)

[The Unique Constraint](#)

[The PRIMARY KEY Constraint](#)

[The FOREIGN KEY Constraint](#)

[The MATCH Part](#)

[The Referential Action](#)

[The CHECK Constraint](#)

[Defining the Assertion](#)

[Using the Domain Constraint](#)

[Chapter 6:](#)

[Creating an SQL View](#)

[Adding a View to the Database](#)

[Defining the View](#)

[Creating an Updatable View](#)

[How to Drop a View](#)

[Database Security](#)

[The Security Scheme](#)

[Creating and Deleting a Role](#)

[How to Assign and Revoke a Privilege](#)

[Chapter 7:](#)

[Database Setup](#)

[Creating a Database](#)

[Deleting a Database](#)

[Schema Creation](#)

[Specific Software Considerations](#)

[Creating Tables and Inserting Data](#)

[How to Create a Table](#)

[Creating a New Table Based on Existing Tables](#)

[How to Insert Data into a Table](#)

[Inserting New Data](#)

[Inserting Data into Specific Columns](#)

[Inserting NULL Values](#)

[Chapter 8:](#)

[Table Manipulation](#)

[Altering Column Attributes](#)

[Renaming Columns](#)

[Deleting a Column](#)

[Adding a New Column](#)

[Alter a Column without Modifying the Name](#)

[Using ALTER TABLE and a Few Rules](#)

[Chapter 9:](#)

[Time](#)

[Datetime Data Types](#)

[Time Periods](#)

[Time Period Tables](#)

[System Versioned Tables](#)

[Chapter 10:](#)

[Database Administration](#)

[Recovery Models](#)

[Database Backup Methods](#)

[Restoring a Database](#)

[Restore Types](#)

[Attaching and Detaching Databases](#)

[Detaching the Database](#)

[Attaching Databases](#)

[Chapter 11:](#)

[Logins, Users and Roles](#)

[Server Logins](#)

[Server Roles](#)

[Assigning Server Roles](#)

[Database Users and Roles](#)

[Assigning Database Roles](#)

[The LIKE Clause](#)

[The COUNT Function](#)

[The AVG Function](#)

[The ROUND Function](#)

[The SUM Function](#)

[The MAX\(\) Function](#)



[The MIN\(\) Function](#)

[Chapter 12:](#)

[Dealing with Errors](#)

[SQLSTATE](#)

[The WHENEVER Clause](#)

[Diagnostics](#)

[Exceptions](#)

[Conclusion](#)

[The Next Steps](#)

[References](#)

# INTRODUCTION

SQL, which stands for Structured Query Language, is a particular programming language that's focused on managing and manipulating databases instead of writing software. After being in development for decades, this language entered the mainstream scene and became the preferred language in data mining, database management, and combined with object-oriented programming languages like C++, it can be used to create complex business applications.

Computers store data in a categorized fashion. Our hard drives are host to many files, whether they're text files, sound files, or image files. Accessing and manipulating such diverse data can't be done efficiently without some kind of system that can bring everything together. This is where the database comes in.

A database can be described as a collection of data that's stored on a computer in such a way for us to easily access, manipulate, and manage it. They work by storing all of our information in an assemblage of tables. This is where SQL comes in handy, and with the help of this book, you're going to explore its most important features.

Databases and database management systems are important for companies, government agencies, and many other organizations. SQL experts are always needed, and a lot of people shy away from such work, thinking that you need to be gifted to manage a database. That's not the case at all, and this book seeks to lift the veil of mystery by guiding you through the foundation of a database, the core elements of SQL, and teach you how to work with data. The goal of this book is to offer you the tools you need to create a relational database and use SQL to manipulate and manage its data.

# CHAPTER 1:

## Understanding Databases

Before the age of computers, we used typewriters and various mechanical calculators to record data and perform mathematical operations with it. That data was then stored in massive rooms packed with cabinets and boxes with paperwork. Nowadays, we don't have to rely on limiting physical space to keep track of information. We have computers and databases.

To keep all of this data safe, however, we need to prepare:

- A good data storage system has to operate quickly because data is regularly accessed, modified, and used in calculations.
- Data storage security is key. Without making sure that data will be safe and reliable in the future, there is a huge risk. Data loss is often due to human or system errors, and can be easily avoided by using backups and frequent hardware and software checkups.
- We need to consider our data filtering methods and tools because in a professional environment, we might have hundreds of terabytes worth of information and we wouldn't be able to manually sort through it all.

Fortunately, modern SQL databases allow us full control over them through data management systems and syntax. Through SQL we can create and manage databases that contain anywhere from, say, twenty data items to thousands. But before we go into the technical details, you need to understand what a database is, what types there are, and how a data management system works. If you already know the basics and you're

interested only in pure SQL, feel free to skip to the second chapter right away.

## What's a Database?

The meaning of the term has changed over time from the days when people defined a database as a collection of data objects inside a list, to nowadays when we look at it as a repository of structured records that we can access through a management system and SQL. Take note that when we talk about records, we refer to a data item's representation.

What does that mean? Imagine a business that keeps track of all their data. Each record can represent a client, and the record will contain all of the client's attributes like their ID, address, phone number, and similar details.

However, a database isn't meant only for conventional data we work with directly. It also stores metadata. This type of data contains information about the data in our database and tells us how it's structured. This is a crucial aspect because without this information, it would be much harder to gain access to our data and change it as needed. In more technical terms, this means an SQL database is self-describing because it tells us everything we need to know regarding the relations between the data items inside and what they're made of. We can find the metadata in a data dictionary. The data dictionary is used to describe all elements that make up the database.

Databases come in all shapes and sizes depending on their purpose, but generally we can place them all into three categories:

1. The smallest and simplest type of database is the **Personal Database**. It's used to store a user's information, doesn't contain vast numbers of data items, and the structure has a basic form.
2. Next, we have the **Group Database**. This category is more complex because it's meant to serve groups of people, such as teams of developers or a company's department. A more unique characteristic compared to the personal database is that this type of database has to be accessed from multiple access points at the same time.
3. Last but not least, we have **Enterprise Databases**. As you can imagine, they're highly complex, deal with vast amounts of data that could never fit on a personal computer, and they require special maintenance on a regular basis.

This is all it takes to figure out what kind of database you're dealing with. Look at its size, the number of users it serves, and the level of tech it requires to run. Now the question is, how do we manage these databases?

## Database Management Systems

For databases to be managed, we require a database management system. Remember that a database is a structure that contains data, but we still need a tool to build that structure and customize it. There are quite a few options on the market, and it all depends on personal preference and your project requirements, because these systems can be wildly different from one another. For example, certain management systems are specifically designed to work with enterprise databases and manage massive amounts of data. Others are for personal use and thus are intended for small amounts of data and personal computers. You don't want to use this kind of system on enterprise-grade data or something will start smoking sooner or later.

Another important thing to consider is the use of the cloud. Sometimes information has to be uploaded to the cloud, or extracted from it to a local machine. In this case we need to use special cloud services and tools that enable us to manage our database systems remotely.

With that being said, here are some of the most popular database management systems: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, Microsoft Access, MongoDB, and many more. Make sure to briefly read about some of the more well-known databases. Many of them are used professionally and some of them are free to download. For study purposes, it doesn't really matter which one you choose. You won't be working with enterprise-level databases just yet.

## Flat Files

The most basic type of data structure you can use is the flat file. As the name suggests, it's a simple structure that's essentially just a list of data objects without any metadata. Here's an example:

John Johnson	233 S. Rose Lane	Anaheim
Jane Moriarty	2331 Newton Lane	Durham
Philip Masterson	123 Aberdeen Avenue	Aberdeen
Sam Cromwell	2345 James Street	Oxford

George Smith	111 Newton Lane	Inverness
Rob Robins	2358 New Lane	Birmingham
Julian Crane	989 Smith's Road	Santa Ana
Mary Jameson	56 Crane's Road	Canterbury

Our flat file is simple and contains only basic information. We do have control over certain attributes like determining how many characters we can input in each field. This means that to read and manipulate the file, we need a program that's capable of accessing and manipulating flat files because it needs to identify each data field. Take note that here we aren't technically dealing with a database. The fields are processed separately and they're read directly. This means working with flat files doesn't take long at all and they can be a useful data storage tool, however, there are some downsides to consider.

Flat files are only optimal when it comes to storing small amounts of data. For instance, if you're familiar with Linux or any other Unix-type operating system, you'll find flat files such as the "passwd" and "group" files. So, they are used in real situations. However, when we have a complex system, a database is needed, even though the processing time will be longer. A database is a lot more versatile because it can be as small or as large as needed.

Furthermore, if you're a software developer, you'll want to work with databases instead of flat files because they're easier to manage and you have more options. You won't even need to know the intricacies of the database structure because the management system will point you in the right direction and perform certain tasks automatically.

## Database Types

Over the decades, databases went through many evolutionary changes. They started out having a hierarchical structure, but this proved to cause certain issues because they weren't very flexible. We need to be able to manipulate data at all times because static databases aren't always useful. This structural problem paved the way to developing a network-focused database. The structure was polished as a result and flexibility was allowed. But with this evolution, another problem arose. This type of database was extremely

complicated and thus posed a new set of challenges for database engineers and analysts.

Fortunately, the *relational database* was introduced afterwards, with its simple yet flexible structure. This is when SQL came to the scene and made all other database models obsolete.

## **The Relational Database**

Relational databases appeared in the 70s and started being used on a wide scale in the 80s. This is also the time when the Oracle Database Management System was developed. Relational databases took the world by storm because we were able to manipulate the structure of the database and thus be highly flexible with the programs we developed as well. Keep in mind that while relational databases are decades old, they are still being used! Other database models are on the rise, but they don't always fulfill the same role as the relational database. For instance, when it comes to big data, relational databases are still mainstream.

One of the advantages of working with this type of database is that you can create two tables in the same database and insert data in both of them, without necessarily connecting them. This means we can freely change the data in one table without worrying that we're going to ruin the data in the other table.

So, how does a relational database work? Imagine a family dinner. You are connected to everyone at the table (you have a relation), but everyone has their own unique characteristics that aren't shared with you. So, the relational model has tables with relations between them, and each table has elements that are unique and not connected to any other data table.

Another important part of databases, which we'll thoroughly discuss in a later chapter, is the view. Views allow us to visualize all the data tables we create. We can always call them up, and therefore they don't have to display all the data inside our database. We can construct a view to show only certain columns and row, and we can even give rights only to certain users to do so. Views are very useful when we need to present only the information we're interested in. Keep in mind that whatever you store in a large database will



contain data objects that may become relevant only in years to come. So, this is an important tool to have when presenting data or analyzing it yourself.

Views are an integral part of databases. You'll sometimes see them referred to as virtual tables. We can stitch them up however we want depending on our needs. We can even take data from multiple tables that have no connection to each other and combine it in one view. However, views aren't as "physical" as tables and databases, because all they do is provide us with a visual representation of the data. The columns and rows in tables contain the data itself, so if we manipulate a view, we only change the data that's displayed, not the data that's actually stored. Imagine a database containing a "customer" data table, and an "invoice" one. The first will store information about the customer, like their name and address. The other table is meant to store the ID number of the invoice and the payment system used by the customer. Now, if your manager comes looking for some information on a customer, he may be interested only in their name and phone number and nothing else. Will you show him the entire database and manually sift through it? Of course not. You will use a view instead, and only display the table of interest or even just the particular columns that we need, like the customer name column.

Views are part of the entire process of working with databases and we couldn't survive without them.

Now, let's discuss the main elements that form the structure of the database, and by this we aren't referring to tables, columns, and rows. The data of a database is managed inside a structure that is made out of a schema, domain, and a set of constraints. The purpose of the schema is to describe and process the structural arrangement of the tables, while the domain dictates the type of data values that can be stored. Lastly, we can have a set of constraints that can place various limits on the data that can be inserted or the user who has the ability to make changes. Here are a few more details on each element (we'll go through them in the next chapters as well):

- **The Schema:** This component is in some ways the structure of the database and it's also what contains the metadata. As mentioned before, the metadata is important because it gives us all the

information we need about the structure of the database.

- **The Domain:** Columns have attributes, and each one has a specific domain with a value that represents it. For example, if you have a database that contains all of the company's cars, you might have a color column that will contain all the colors as values. The color is an attribute and the distinct color values represent the domain of the attribute. In other words, a data domain simply involves only the values that are valid for the column.
- **Data Constraints:** This component is sometimes optional, however, it's important for real world database management. A lot of beginners don't work with constraints at first, but as you'll learn very soon, they are necessary. The purpose of a constraint is to specify the type of value that can be linked to an attribute. Furthermore, it allows us to place access or data manipulation restrictions on certain users.

You don't want to give permission to just anyone so that they can make database modifications. Data could be erased on purpose or modified by accident, and we need to avoid that. Therefore, constraints act as barriers or restrictions on data or entire tables. So, in a way, we can say that a column's domain is not determined just by the data values, but also by the constraints we implement. For instance, in our car color example, we can apply a constraint to the column to accept a maximum of five values. If we already have car colors all set, we won't allow another user to add more. As you can see, having ways to limit the access to our data can be quite handy. Otherwise we run the risk of someone corrupting our information by introducing values that might not actually exist or are just plain wrong. This erroneous information will then affect any analysis or any other data we might pass on to other departments and influence them negatively from the start.

All the points made above should tell you why relational databases are still being used today even though new models were created. While they don't serve every scenario, you will encounter them if you choose to work with databases professionally. Relational databases aren't Jacks-of-all-trades though, because they are in some ways limited. For instance, in recent times, hardware evolution has allowed programming languages to become a lot

more versatile and powerful. This showed us that a lot more can be done with complex notions like inheritance and encapsulation. Unfortunately, relational databases can't take advantage of some of these features, and that's why new models were developed.

Object-oriented programming is what drives most of our software and computer games today, so a new database model referred to as the object model was created. However, chances are you won't work with this particular model nowadays because it came with a different set of problems. Fortunately, shortly after its development, it was combined with the relational model to form an improved version that is now used pretty much everywhere, whether for personal reasons or corporate. This model is known as the object-relational model, and it borrows multiple features from the relational database we just discussed.

## **The Object-Relational Database**

The relational model brought a lot of advantages, and with the mainstream use of object-oriented programming languages, the object model came to be. However, the real advancement was made when the two models combined. Now, we have the versatility and compatibility that comes with using relational databases combined with the benefits brought by the object model, which added a whole new set of functionalities.

SQL is at the heart of implementing the object-oriented features into the relational model. That's why we have database management systems that can handle object-relational models just as well as they can manage relational models. This is all due to the compatibility between the two models and the smooth transition allowed by SQL. The relational database model is still at the core of what we're working with today, but the new model is regularly being expanded to further integrate object-oriented principles.

In this book we are going to mainly focus on the object-relational model, however, by mastering its concepts you should still learn more than enough to be able to work with the older but still used relational model.

# CHAPTER 2:

## Using Queries to Obtain Data

When working with databases, we rely on queries to request and retrieve information from our data tables. The information is then generated as a textual result, or even a graphical one if, for example, we're analyzing various trends from data obtained through data mining.

There are multiple query languages we can use to work with the information stored in data tables, but we're working with SQL because it's the most commonly used one by data admins. SQL can process a wide array of queries, anywhere from the most basic SELECT queries to the more complex ones. In fact, many beginner SQL students are shocked to learn that other query languages exist. Even though SQL is the most common one, there are alternatives like LINQ, if you choose to spend some time exploring. However, throughout this book you'll be using SQL queries to retrieve or manipulate data. Most query language alternatives are in fact very close to SQL, just like Java is similar to C#.

The main difference is that each query language will generate certain data types based on the function. For instance, SQL will display your data in simple rows and columns, just like Microsoft Excel does. Other query languages might be more specialized in generating graphs, complex data analysis charts, and much more.

With that being said, in this chapter we're going to focus on queries because they're going to be used throughout the book. Understanding basic query flow is crucial in order to find solutions to the most common problems you'll

encounter when working with data.

## The Basics

The best way to imagine query flow is by looking at it in a logical order, like this: SELECT > FROM > WHERE > GROUP BY > HAVING > ORDER BY. SELECT can be used on its own as a statement to retrieve data from any tables or specific rows and columns. However, it's important to remember the order in which the clauses are processed. Here's a simple example of the entire query with all the clauses included:

```
SELECT  
[DISTINCT] [TOP (n)] *, columns, or expressions  
[FROM data source]  
[JOIN data source  
ON condition]  
[WHERE conditions]  
[GROUP BY columns]  
[HAVING conditions]  
[ORDER BY columns]:
```

First, we initiate the SELECT statement which has to include at least one expression, but it can contain a whole list of them and columns as well. For instance, you can use a simple integer or string as the expressions. Here's the most basic SELECT statement you can probably have:

```
SELECT 1;
```

Next, we have the FROM clause that's used to compile the sources of information into a result that we can use. These data sources will usually appear as tables. We can also add a JOIN clause to reference several tables.

The WHERE clause is then used to filter certain rows from the data source we obtained. The filtering action is performed based on a set of conditions.

Afterwards, we introduce a GROUP BY clause to create smaller sets of information from a larger one by defining which columns we want to focus on.

Optionally, we can use the HAVING clause as well to add a restriction to the results gained from the aggregation.

Finally, we have the ORDER BY clause so that we can specify in which

order to sort our results. We can have ascending or descending results, depending on the columns we specify in the **SELECT** statement's column list. You can skip this clause if you want because by default, all the results will be presented in an ascending order.

## Query Structure and the **SELECT** Statement

Now, for our next few practical examples and demonstrations, we are going to use the following Product table.

<b>ProductID</b>	<b>Name</b>	<b>ProductNumber</b>	<b>Color</b>	<b>StandardCost</b>	<b>ListPrice</b>
317	LL Crankarm	CA-5965	Black	0	0
318	ML Crankarm	CA-6738	Black	0	0
19	HL Crankarm	CA-7457	Black	0	0
320	Chainring Bolts	CB-2903	Silver	0	0
321	Chainring Nuts	CN-6137	Silver	0	0

It's important to understand the syntax and the structure of a query, so let's start with a couple of examples using the **SELECT** statement in more detail to learn how it's used to extract information:

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name
```

In this example, we chose two columns and every row in the table. Take note that by selecting two columns at the same time, the query will execute faster than when making your selection this way:

```
SELECT *  
FROM Table_Name
```

Now, continue on your own by using the same table, but now select the name, color, product number, and product ID columns. Just remember that there are no spaces within column names.

## The WHERE Clause

When a query returns a number of rows, you might want to filter through them by using the **WHERE** keyword. This is a clause that relies on one condition. For instance, we can use it when the column is less than, equal to, or greater than a given value.

Remember that the **WHERE** clause follows the **FROM** statement when you write your code. In addition, the operators you can use will depend on the data types you need to filter.

Furthermore, you can use **EQUALS** to find the perfect match. In the next example, we are going to use the **WHERE** clause to filter through a column that stores a 'Value' string. Take note that strings are written with single quotes, while values are written without them.

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 = 'Value'
```

Next, we might use the **BETWEEN** keyword to look for a value that's between a range of numerical or datetime values. You'll notice that when using this comparison operator, we'll have a lower first value (on the left), and higher second value (on the right). Here's an example where we make such a comparison:

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 BETWEEN 100 AND 1000
```

Next, we have the collection of **GREATER THAN**, **LESS THAN**, **LESS THAN OR EQUAL TO**, and **GREATER THAN OR EQUAL TO** comparison operators, which we can use to compare our values:

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 =< 1000
```

**LIKE** is another operator that we'll use only when looking for values and strings found inside a column. Take note that strings are still written in between single quotation marks; we'll also have to add the percent symbol to specify whether the string is found at the start, the end, or somewhere in between the range of strings. Here's how this operator is used in an example:

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 LIKE '%Value%'
—Searching for the string 'value' in every field
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 LIKE 'Value%'
—Searching for the string 'value' at the beginning of the field
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 LIKE '%Value'
—Searching for the string 'value' at the end.
```

Another pair of operators you'll be using are the **IS NULL/IS NOT NULL** pair. We mentioned earlier that **NULL** is just a cell without data. In the real world you'll encounter tables with null values and you need to deal with them:

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 IS NULL
```

Here we filtered out all values that are not **NULL** in the column.

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 IS NOT NULL
```

These lines will filter out any value that is **NULL**.

To get some practice, go back to the Product table, select the ID, name, number and color, and start filtering only the items that are silver in color.

## Using **ORDER BY**

The next clause on our list is the **ORDER BY** clause which is needed to sort data in ascending or descending order. It also allows us to specify the column by which we want to sort. Take note that by default, the clause will sort all the data in ascending order. If you need it sorted in descending order, all you need to do is add the **DESC** keyword. Here's how it works:

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 = 'Value'
ORDER BY Column_Name1, Column_Name2
```

Everything is sorted in ascending order. You can also add the **ASC** keyword to make the process more obvious.



```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 = 'Value'  
ORDER BY Column_Name1, Column_Name2 DESC
```

In this example we sorted the information in descending order because we added the **DESC** keyword at the end of the list of columns.

Practice using these examples with the data from the Product table. Select the columns and sort the prices within a certain price range from the lowest to highest. Take note that the \$ will not be used in your query.

## Subqueries

This concept simply refers to introducing a subquery inside another main query so that we can execute a logic test or a mathematical operation, giving a result that is then used by the main query. Writing subqueries isn't difficult and mastering them will increase the flexibility of your statement, as you'll be able to perform more complex processes.

In the real-world, subqueries are used to retrieve a number of data objects and then work with them using a **FROM** statement that is part of the main query. Using this concept, we can also run scalar subqueries to return a value and then process it with a statement that filters throughout the data rows. Both of these scenarios are where you'll encounter subqueries most often.

Now, let's look at a practical example of how to apply a subquery to a table update. Take note that we'll use our data and all defined conditions to create the subqueries that will be used to look for the values that match with certain columns.

```
UPDATE table_one  
SET thiscolumn = (SELECT thiscolumn  
FROM table_two WHERE table_one.thiscolumn = table_two.thiscolumn)  
WHERE EXISTS (SELECT thiscolumn  
FROM table_two WHERE table_one.thiscolumn = table_two.thiscolumn) ;
```

As you can see, we wrote two subqueries following an identical syntax. The **SELECT** statement is the first subquery that's part of the **SET** statement. Its purpose is to return the values that we need to update the table. The second subquery is then used as part of the **WHERE EXISTS** statement, where we again use a **SELECT** statement as well because we need to filter through the

information to find out what needs to be updated.

Take note that the subqueries in our examples are known as *correlated subqueries*, which means they are connected to a certain element that's part of the main query. In this case that element is the first table found inside the update process. On the other side, we can also have uncorrelated subqueries, which would mean that they don't have any component dependence inside the main query.

## Filtering Data with Subqueries

The first thing that comes to mind regarding data filtering operations is using the **WHERE** statement. You should already be familiar with it because it's often used to form a conditional statement like **WHERE price > 150**. The problem with this is that you have to already have that value of "150" in order to perform the operation. In the real world, you will sometimes not have that luxury, so you need to figure out a way around the problem. This is where you start filtering your data by using subqueries. For instance, you can create a set of values through a subquery and then apply the **WHERE** statement.

With that in mind, let's say we need to learn which are the largest European cities by population. Assuming we want to discover all the cities that contain 10% of the population, we can focus on determining the 90<sup>th</sup> percentile in the rest, and then run a data filtering operation inside a subquery to get the results we need. Here's how the code looks:

```
SELECT city_name
       city_eu
       p0010001
FROM cities_eu_2008
WHERE p0010001 >= (
    SELECT percentile_cont (.9) WITHIN GROUP (ORDER BY p0010001)
    FROM cities_eu_2008)
ORDER BY p0010001 DESC;
```

You should already understand the query, however, in this example we also use the **WHERE** statement to filter through a column's data, but the result doesn't return the value we're looking for. That's why we use a percentile function to generate the value needed for this process, and then use it inside the main query. With that being said, you should remember that subqueries are only practical inside a **SELECT** or **DELETE** query, especially when using the

**WHERE** statement as well. Yes, you can use the methods in this section to also delete something from the table, however, that would be a bit more complicated than the usual methods you have already learned, and not as efficient. The only case where it would be truly useful is when working with a massive database that contains millions of items which you would then split into manageable parts. Filtering through such large amounts of data by using subqueries would take a lot of time otherwise.

## Derived Tables with Subqueries

If you have a subquery that yields a number of results, you can convert it to a data table by inserting it inside a **FROM** statement. Remember that all data is just a set of rows and columns, and therefore we can use any information we retrieve to generate what's known as a *derived table*.

Derived tables are actually exactly the same as your typical tables, and you can alter and manager them exactly the same way by using the SQL operations you learned so far. So, what makes them special? Why would you use them at all if they're the same?

Imagine you have a number of processes and mathematical operations to execute, and obviously you can't just use a single query to deal with everything. That's where the concept of a derived table can come in handy. Just think of how a median value can be a more accurate representation than an average, which is susceptible to outliers. Due to such problems, you would actually calculate both values and then compare them to see whether you have a well-distributed set of data. However, if you learn that there's a big difference between the two results, you know you have problems with outliers. So, let's use subqueries using our previous city example and find those values and compare them. In addition, we are going to use the derived table concept with a subquery in the **FROM** statement:

```
SELECT round(calcs.average, 0) AS average,
       calcs.median,
       round (calcs.average - calcs.median, 0 ) AS median_average_diff
FROM (
  SELECT avg (p0010001) AS average,
         percentile_cont(.5)
         WITHIN GROUP (ORDER BY p0010001) :: numeric (10, 1)
         AS median from cities_eu_2008
)
```

```
AS calcs;
```

The example should be quite clear. We use the percentile function once again, together with the average function, and we determine the two values based on the population numbers. Then the subquery is referenced as a derivative table, and the values are returned to the main query. From your final result you should deduce that there's a big difference between the average and the median values, and therefore we can conclude that there are a few very large cities that lead to an increase to the average value. These few cities are the outliers in our data.

## Table Expressions

Creating derived tables using **FROM** statement subqueries isn't the only method. We can develop these tables through a common table expression (CTE). This method enables us to create a derived table by writing subqueries that rely on the **WITH** statement. Once data is returned this way, we can use other queries to manipulate it further because the subquery comes before the main query in this case.

Let's use the same cities example to create an example that relies on the common table expression. Imagine having to figure out how many cities have more than 500,000 people living in them. Here's how the code looks:

```
WITH  
  large_cities (city_name, st, p0010001)  
AS (  
  SELECT city_name, eu_city, p0010001  
  FROM cities_eu_2008  
  WHERE p0010001 >= 500000 )  
SELECT st, count (*)  
FROM large_cities  
GROUP BY st  
ORDER BY count (*) DESC ;
```

Now let's see what happened. Take note that we started a section using the **WITH AS** statement to create the new table called **large\_cities**. Afterward, we returned the data columns that are part of the table. Pay attention to this process because it's not identical to working with the **CREATE TABLE** statement. The main difference is that we don't need to define our data types because they're automatically inherited from the subquery that comes right

after the `AS` command.

The new columns part of the large cities table is returned by the subquery, but that doesn't mean that they have to have matching names. In fact, the columns aren't even needed if we don't change their names. The reason why we're calling for the column list is because that way we can see the whole picture, and we can analyze the data we're dealing with to understand the process. Finally, our parent query contains every object that's part of the table and groups them together. The end result should be a list of cities that goes from the most populated one to the least.

If you don't like this method, remember that you can also use the `SELECT` statement instead of the CTE. However, the purpose of this technique really starts shining when working with large databases and not tiny examples like the ones we're playing with. When it comes to large datasets, the common table expression enables us to examine just pieces of the whole instead of processing the entire mass of information at once. Afterwards, we can add that data to a main query.

In addition, all the tables you define by using this method can be used as segments of the main query. Just take note that if you use the `SELECT` statement, you need to declare it whenever it needs to be used. Therefore, all the code that relies on the CTE will be more efficient and easier to read as well as process. At this point, you might not see the practical use of this technique, but keep this information in the back of your mind. One day you'll need it.

## Cross Tabulations

Being able to summarize all the values in a neat, easy-to-read table is an important tool when analyzing your data. That's what cross tabulations are for. They are tables that are actually more similar to a matrix, and they're used to easily make data comparisons. If you aren't familiar with the concept of a matrix, imagine have a variable inside a row and then another that's represented by a column. Where the two variables meet, or intersect, we have a value. As you can see, this clue is found in the term "cross tabulation." Practically, you might use them when summarizing reports, for instance a political report like in this simple example:

Candidate	District 1	District 2	District 3
Smith	666	2211	3784
Jameson	898	198	1656
Woods	767	987	878

Our two row and column variables are the political candidate and the district. Values can be found in each cell where the row intersects with the column. Now, let's talk more about actual cross tabulations.

By default, SQL isn't capable of generating cross tabulations because it lacks the functions for it. So, the solution is to work with a well-developed database management system like *PostgreSQL*, because we can find modules for it that can extend its usability through non-standard SQL functions and features. Take note that you can't use the same module for every database management system. If you're using this one, then you can install the *tablefunc* package to get the cross tabulations feature. If you use other system, you need to research for potential modules, if the system even allows such features. Before we get to our actual cross tabulation, here's how the module is installed in *PostgreSQL*:

```
CREATE EXTENSION tablefunc;
```

The module will now be installed and you'll be able to work with the syntax needed for cross tabulations. We'll go over it in the following section.

## Tabulating

Let's say you're part of a corporation that likes to set up all sorts of fun activities for team building during certain weekends. Such activities are often planned between different offices or departments to stoke a bit of competition as well, but the employees can't agree on which activity to go for. So, we need to analyze some data to figure out which activity is the most preferred by the majority. Let's say we'll conduct a survey with 200 participants. We'll have a **participantID** row, **officebranch**, and **funactivity** rows as well. Based on the data we collect and analyze, we will present a readable result that can be added to a simple report for the company's executive. Here's how this scenario would play out in SQL:

```
CREATE TABLE activitychoice (  
  participantID integer PRIMARY KEY,  
  officebranch varchar (30),  
  funactivity varchar (30) );  
COPY activitychoice  
FROM 'C: |MyFolder|activitychoice.csv'  
WITH (FORMAT CSV, HEADER);
```

Take note that the .CSV file is the survey we conducted on the 200 participants. So, all of the results are stored in a basic Excel-like spreadsheet. Let's continue with the exercise and see what kind of results we could get:

```
SELECT *  
FROM activitychoice  
LIMIT 5;
```

For the sake of this example, we're going to pretend that most participants from the five office branches chose paintball. Next, we need to generate the cross tabulation like so:

```
SELECT *  
FROM crosstab ('SELECT officebranch, funactivity, count(*)  
  FROM activitychoice  
  GROUP BY officebranch, funactivity  
  ORDER BY officebranch,  
  'SELECT funactivity  
  FROM activitychoice  
  GROUP BY funactivity  
  ORDER BY funactivity')  
AS (officebranch varchar (30),  
  paintbal bigint,  
  archery bigint,  
  football bigint);
```

As you can see, we have a **SELECT** statement that is used to return the contents from the cross tabulation function. Afterwards, we execute two subqueries inside that function, with the first one generating its own data by taking the input from three data columns (**officebranch** that has data on the office locations, **funactivities** which has data on the preset activities, and lastly the column that contains the intersecting values).

In order to return the number of the participants that chose a certain activity, we need to cross the data objects. To do that, we execute a subquery to generate a list in which another subquery will insert the categories. The

crosstab function is then used to instruct the second subquery to only return a single column on which we apply the `SELECT` statement. That's how we gain access to all the activities in the list so that we can group them and return the values.

You'll notice that we're also using the `AS` keyword. In this example we need it to specify the data types inside our cross-table's data columns. Just make sure you have a match for the names, otherwise the subqueries won't work properly. So, if the subquery returns the activities in a certain order, the result column must apply the same rules. Here's how our theoretical end-result would look:

<b>officebranch</b>	<b>paintball</b>	<b>archery</b>	<b>football</b>
Central	19	29	23
Uptown	47		17
Downfield	20	14	24

The data is now readable, and we can add it to a simple report that can be shown to the manager. We can see which activity is the preferred one, but we also spotted a null value in the second column. This simply means that nobody from Uptown chose archery.



# CHAPTER 3:

## The Data Definition Language (DDL)

The data definition language is needed to define, delete, or manipulate databases and the data structures within them through the use of the create, alter, and drop keywords. In this chapter we are going to discuss each aspect of SQL's DDL.

### Using DDL to Create

As mentioned before, the DDL is needed to build databases in the management system we use. Here's how the syntax looks for this process:

```
CREATE DATABASE my_Database
```

For instance, when we set up a database containing some information such as "customer details," we are going to write the following statement:

```
CREATE DATABASE customer_details
```

Don't forget about the use of uppercase or lowercase because SQL is a case sensitive language. With that in mind, let's set up a number of customer tables that will contain all of our data on our customers that's stored in the database we just created. This is a perfect example of working with a relational database management system since all of our data is linked and therefore accessing or managing our information is an easy task. Now, let's use the following code:

```
CREATE TABLE my_table  
(  
  table_column-1 data_type,
```

```

table_column-2 data_type,
table_column-3 data_type,
...
table_column-n data_type
)
CREATE TABLE customer_accounts
(
acct_no      INTEGER, PRIMARY KEY,
acct_bal     DOUBLE,
acct_type    INTEGER,
acct_opening_date DATE
)

```

Take note of the primary key attribute. Its purpose is to guarantee that the “acct\_no” column will contain only unique values. In addition, we won’t have any null values. While we should have a primary key field for each record, other attributes will be “not null” in order to make sure that the columns won’t take in any null values. Furthermore, we also need a “foreign key” so that the linked data record from other tables won’t be removed by accident. The column with this attribute is actually just a copy of the primary key from the other table. So, for instance, in order to create a new table like “customer\_personal\_info” inside our database, we can type the following lines:

```

CREATE TABLE customer_personal_info
(
cust_id      INTEGER PRIMARY KEY,
first_name   VARCHAR(100) NOT NULL,
second_name  VARCHAR(100),
lastname     VARCHAR(100) NOT NULL,
sex          VARCHAR(5),
date_of_birth DATE,
address      VARCHAR(200)
)

```

As you can see, in this new table we have a **cust\_id** primary key column. The **customer\_accounts** table will also have to include a **cust\_id** column in order to make the connection with this table. This is how we gain customer data by using only an account number. So, the ideal method of ensuring the integrity of information added to the two tables is to add a **cust\_id** key in the form of a foreign key to the **customer\_accounts** data table. This way the data from one table that’s related to that of another table can’t be removed by accident.

## Adding Foreign Keys with ALTER

We created a new table, so now we need to change the other table in order to include the foreign key. Here's the DDL syntax we need to use:

```
ALTER TABLE mytable  
ADD FOREIGN KEY (targeted_column)  
REFERENCES related_table (related_column)
```

Next, we need to write the following statements in order to include the key to the **customer\_accounts** table and reference it to the **cust\_id** from the other table.

```
ALTER TABLE customer_accounts  
ADD FOREIGN KEY (cust_id)  
REFERENCES customer_personal_info(cust_id)
```

## Creating Foreign Key DDL

In some scenarios we'll have to create foreign keys when we set up a new table. Here's how we can do that:

```
CREATE TABLE my_table  
(  
Column-1 data_type FOREIGN KEY, REFERENCES (related column)  
)
```

## Unique Constraints

In order to make sure that the data we store in the column is unique like that from the primary column, we need to apply a unique constraint. Keep in mind that we can have an unlimited number of unique columns, however, we can have only one primary key column. With that in mind, here's the syntax used to set up a unique column:

```
CREATE TABLE my_table  
(  
Column-1 data_type UNIQUE  
)
```

## Deleting Tables and Databases

```
DROP TABLE my_table
```

Take note that you can't go back once you perform this action. So, make sure you really need to remove a table before you use this query. Now, here's the

syntax used to delete the entire database:

```
DROP DATABASE my_database
```

Just as before, you can't reverse this process. Once you delete the database, it's gone. On a side note, you might want to just remove a column from the table. In that case, we are going to use the **DELETE** statement like so:

```
DELETE column_name FROM data_table
```

## How to Create Views

```
CREATE VIEW virtual_table AS  
SELECT column-1, column-2, ..., column-n  
FROM data_table  
WHERE column-n operator value
```

Here's an example using the **customer\_personal\_info** table:

<b>cust_id</b>	<b>first_nm</b>	<b>second_nm</b>	<b>lastname</b>	<b>sex</b>	<b>Date_of_birth</b>
03051	Mary	Ellen	Brown	Female	1980-10-19
03231	Juan	John	Maslow	Female	1978-11-18
03146	John	Ken	Pascal	Male	1983-07-12
03347	Alan	Lucas	Basal	Male	1975-10-09

Table 2.1 customer\_personal\_info

Now, let's say we want to create a view of only the female customers. Here's what we need to do:

```
CREATE VIEW [Female Depositors] AS  
SELECT cust_id, first_nm, second_nm, lastname, sex, date_of_birth  
FROM customer_personal_info  
WHERE sex = 'Female' 2
```

Next, we can declare the following statements in order to see the data from the view we created:

```
SELECT * FROM [Female Depositors]
```

This is the result we would get:

<b>cust_id</b>	<b>first_nm</b>	<b>second_nm</b>	<b>lastname</b>	<b>sex</b>	<b>Date_of_birth</b>
----------------	-----------------	------------------	-----------------	------------	----------------------

03051	Mary	Ellen	Brown	Female	1980-10-19
03231	Juan		Maslow	Female	1978-11-18

Table 2.2 view of customer\_personal\_info

Take note that views are recreated only as needed and they are never stored inside the computer memory. However, the view is presented exactly like a table.

# CHAPTER 4:

## SQL Joins and Union

We can add logical operators such as **AND** to our select statement in order to process multiple tables in the same statement. Take note that you can also use the join operator (left, right, and inner) for the same process, and you might get a more efficient and better optimized processing result.

### INNER JOIN

The **INNER JOIN** statement will enable you to use a single statement to process multiple tables at the same time. In order for this to work, our tables have to include linked columns, namely a primary column connected to the foreign columns. The **INNER JOIN** operation will extract the data from two or more tables when a relation between the tables is found. Here's how the syntax looks for this process:

```
SELECT column-1, column-2... column-n
FROM data_table-1
INNER JOIN data_table-2
ON data_table-1.keycolumn = data_table-2.foreign_keycolumn
```

So, if you need to extract certain matching data from the two tables, we are going to use the query like so:

acct_no	cust_id	acct_bal	acct_type	acct_opening_date
0411003	03231	2540.33	100	2000-11-04
0412007	03146	10350.02	200	2000-09-13
0412010	03347	7500.00	200	2002-12-05

Table 4.1 customer\_accounts

cust_id	first_nm	second_nm	lastname	sex	Date_of_birth	addr
03051	Mary	Ellen	Brown	Female	1980-10-19	Cover
03231	Juan		Maslow	Female	1978-11-18	York
03146	John	Ken	Pascal	Male	1983-07-12	Liver
03347	Alan		Basal	Male	1975-10-09	Easto

Table 4.2 customer\_personal\_info

```
SELECT a.acct_no, b.first_nm AS first_name, b.lastname AS surname, b.sex, a.acct_bal
FROM customer_accounts AS a
INNER JOIN customer_personal_info AS b
ON b.cust_id = a.cust_id
```

Here's the result we're going to achieve with the above SQL statement in tabular format:

acct_no	first_name	surname	sex	acct_bal
0411003	Juan	Maslow	Female	2540.33
0412007	John	Pascal	Male	10350.02
0412010	Alan	Basal	Male	7500.00

Table 4.3

## RIGHT JOIN

When we use this operator with the **SQL SELECT** query, we can include all the data from the right table without the requirement of any matching data in the left table. Here's the syntax for this process:

```
SELECT column-1, column-2... column-n
FROM left-data_table
RIGHT JOIN right-data_table
ON left-data_table.keyColumn = right-data_table.foreign_keycolumn
```

For instance, if we want to select the information of our customers, such as **acct\_no**, **surname**, and more, across both tables, the **customer\_accounts** and **customer\_personal\_info** will present that data even if there's no active customer account. Here's how our **RIGHT JOIN** statement will look:

```
SELECT a.acct_no, b.first_nm AS first_name, b.lastname AS surname, b.sex, a.acct_bal
FROM customer_accounts AS a
RIGHT JOIN customer_personal_info AS b
ON b.cust_id = a.cust_id2
```

The result of this process can be seen in the following table:

acct_no	first_name	surname	sex	acct_bal
0411003	Juan	Maslow	Female	2540.33
0412007	John	Pascal	Male	10350.02
0412010	Alan	Basal	Male	7500.00
	Mary	Brown	Female	

Table 4.4

## LEFT JOIN

This statement is basically the opposite of the previous statement. **LEFT JOIN** will return the data from the left (first) table, whether we have any matching data or not when comparing it with the right table. Let's take a look at the syntax:

```
SELECT column-1, column-2... column-n
FROM left-data_table
LEFT JOIN right-data_table
ON left-data_table.keyColumn = right-data_table.foreign_keycolumn
```

Now let's say we want to see all the details about the customer's accounts across our two tables. Here's how:

```
SELECT a.acct_no, b.first_nm AS first_name, b.lastname AS surname, b.sex, a.acct_bal
FROM customer_accounts AS a
LEFT JOIN customer_personal_info AS b
ON b.cust_id = a.cust_id
```

And this is the output of our **LEFT JOIN** statement as tabular data.

acct_no	first_name	surname	sex	acct_bal
0411003	Juan	Maslow	Female	2540.33
0412007	John	Pascal	Male	10350.02
0412010	Alan	Basal	Male	7500.00



Table 4.5

## The UNION Statement

Next, we might want to combine our results in a single place. That's what the **UNION** keyword is for, and here's how we can use it:

```
SELECT column-1, column-2... column-n
FROM data_table-1
UNION
SELECT column-1, column-2... column-n
FROM data_table-22
```

Let's say we have a bank that has a number of customers from the US and UK, and they would like to know which ones have an active account in both countries. Here are the two tables with customers from each country, followed by our SQL statement:

<b>acct_no</b>	<b>first_name</b>	<b>surname</b>	<b>sex</b>	<b>acct_bal</b>
0411003	Juan	Maslow	Female	2540.33
0412007	John	Pascal	Male	10350.02
0412010	Alan	Basal	Male	7500.00

Table 4.6 London\_Customers

<b>acct_no</b>	<b>first_name</b>	<b>Surname</b>	<b>sex</b>	<b>acct_bal</b>
0413112	Deborah	Johnson	Female	4500.33
0414304	John	Pascal	Male	13360.53
0414019	Rick	Bright	Male	5500.70
0413014	Authur	Warren	Male	220118.02

Table 4.7 Washington\_Customers

```
SELECT first_name, surname, sex, acct_bal
FROM London_Customers
UNION
SELECT first_name, surname, sex, acct_bal
FROM Washington_Customers2
```

The resulting table will look like this:

<b>first_name</b>	<b>surname</b>	<b>sex</b>	<b>acct_bal</b>
Juan	Maslow	Female	2540.33
John	Pascal	Male	10350.02
Alan	Basal	Male	7500.00
Deborah	Johnson	Female	4500.33
Rick	Bright	Male	5500.70
Authur	Warren	Male	220118.02

Table 4.8 SQL UNION

You'll notice that John Pascal is mentioned once in the resulting table. That's how we know our union statement worked as intended, since this customer is part of both banks.

## The UNION ALL Statement

This SQL statement is nearly identical to the previous union query with the only real difference being that we can see the total number of data records from all the involved tables. Here's how we use it:

```
SELECT column-1, column-2... column-n
FROM data_table-1
UNION
SELECT column-1, column-2... column-n
FROM data_table-2
```

Now we can use this command on our previous customer tables by writing the following SQL statement:

```
SELECT first_name, surname, sex, acct_bal
FROM London_Customers
UNION ALL
SELECT first_name, surname, sex, acct_bal
FROM Washington_Customers
```

Here's the resulting table:

<b>first_name</b>	<b>surname</b>	<b>sex</b>	<b>acct_bal</b>

Juan	Maslow	Female	2540.33
John	Pascal	Male	10350.02
Alan	Basal	Male	7500.00
Deborah	Johnson	Female	4500.33
John	Pascal	Male	13360.53
Rick	Bright	Male	5500.70
Authur	Warren	Male	220118.02

Table 4.9

# CHAPTER 5:

## Data Integrity

Databases are more than just simple data storage units. One of the most important data management aspects is maintaining the data's integrity because if it's compromised, we can no longer trust in the quality of the information. When the information becomes unreliable, the entire database is compromised.

In order to ensure the integrity of the data, SQL provides us with a number of rules and guides that we can use to place a limit on the stored table values. The rules are known as integrity constraints and they can be applied to either columns or tables. In this chapter we are going to focus on this aspect and discuss all the constraint types.

## Integrity Constraints

For a better understanding, database integrity constraints can be divided into three categories:

1. **Assertions:** An assertion definition refers to the definition of an integrity constraint within another definition. In other words, we don't have to specify the assertion in the definition of the data table. In addition, one assertion can be assigned to more than a single table.
2. **Table Related Constraints:** This is a group of integrity constraints that must be defined within the table. Therefore, we can define them in the definition of the table's column, or as a separate element of the table.

3. **Domain Constraints:** These constraints have to be defined in their own definition. This makes them similar to assertions. However, a domain constrain will work on the specific column that's been defined within the domain.

Take note that, of these categories of integrity constraints, the table related constraints come with a number of options. Therefore, they are probably the most popular nowadays. This category can be further divided into two subcategories, namely the **table constraints** and the **column constraints**. Their purpose is self-explanatory, and both of them can function with other integrity constraints as well. On the other hand, the assertions and the domain constraints aren't so versatile because they can only be used with a single type of constraint.

## The Not Null Constraint

Earlier, we discussed that null refers to an undefined value, not the idea that nothing exists. So, take note that this concept is not the same as having default values, empty strings, blank spaces, or zero values. The easiest way to understand the concept of 'null' in SQL is to imagine it as a label that tells us something about the column: that is has an absent value. In other words, an empty column equals a null value and therefore we will be notified that we're dealing with an undefined or unknown value.

It's important to understand this concept, because data columns come with a "nullability" attribute, which alerts us when we're dealing with undefined values. Remember that in SQL we can have null values in the data columns, but if we remove the nullability attribute from the column by using the **NOT NULL** constraint we can make changes to the attribute. We've used this command in other examples and you've seen already that it translates to no longer allowing a column to contain null values.

Remember that in SQL, this constraint must be applied to a column, and therefore we can't use it on the other constrain categories, such as assertions or table-based constraints. With that being said, the syntax is fairly simple because using the **NOT NULL** constraint is an easy process:

```
(name of column) [ (domain) | (data type) ] NOT NULL
```

For instance, let's say that you want to create a new table and you're going to call it **FICTION\_NOVEL\_AUTHORS**. We're going to need to add three columns to this table, namely an **AUTHOR\_ID**, **AUTHOR\_DOB**, and **AUTHOR NAME**. Furthermore, we need to guarantee that the entries we're inserting will contain valid values for the ID and the name columns. This means that we must add the **NOT NULL** constraint. Here's how we're going to accomplish that:

```
CREATE TABLE FICTION_NOVEL_AUTHORS
(
  AUTHOR_ID    INT          NOT NULL ,
  AUTHOR_NAME  CHARACTER(50) NOT NULL ,
  AUTHOR_DOB   CHARACTER(50)
);2
```

Take note that we didn't add the constraint for the **AUTHOR\_DOB** column. So, if there's no value in the new entry, a null value will be inserted by default.

## The Unique Constraint

Take note that in SQL, both table and column integrity constraints will only accept unique constraints. These constraints are divided into two categories: **UNIQUE** and **PRIMARY KEY**.

For now, we're going to discuss mostly the first type of constraint and leave the second for later.

**UNIQUE** is used to deny duplicate values from being inserted in the column. If a value is already in the column, you won't be able to add it again. Now, let's say we need to apply this constraint to the **AUTHOR\_DOB** data column. We'll soon find out that making such a column entirely unique isn't a great idea. So, what do we do? One method is to apply the **UNIQUE** constraint to both the Name and DOB columns because the table won't allow us to repeat the column pair, however, it will let us add duplicate values to the individual columns. The only restriction we have now is the inability to introduce the Name/DOB pair a second time in the table.

Remember that we can use this constraint on the entire table or just a column. Here's how we add it to the column:

```
(name of column) [ (domain) | (data type) ] UNIQUE
```

To add it to the table, we must declare the constraint as a component of the table:

```
{ CONSTRAINT (name of constraint) }  
UNIQUE < (name of column) { [, (name of column) ] ... } >
```

As you can see, this version of the constraint is slightly more difficult to apply. All **UNIQUE** constraints have to be defined. With that being said, here's an example on how we can add it at the column level:

```
CREATE TABLE BOOK_LIBRARY  
(  
  AUTHOR_NAME  CHARACTER (50),  
  BOOK_TITLE   CHARACTER (70) UNIQUE,  
  PUBLISHED_DATE INT  
);2
```

Next, we're going to apply the constraint to other columns as well. However, we will observe different effects than when using the constraint at the table level to apply it on multiple columns. Let's examine this process:

```
CREATE TABLE BOOK_LIBRARY  
(  
  AUTHOR_NAME  CHARACTER (50) ,  
  BOOK_TITLE   CHARACTER (70) ,  
  PUBLISHED_DATE INT,  
  CONSTRAINT UN_AUTHOR_BOOK UNIQUE ( AUTHOR_NAME, BOOK_TITLE )  
);2
```

Now, if we need to insert new entries, the **AUTHOR\_NAME** column together with the **BOOK\_TITLE** column will require unique values. Remember that the purpose of the **UNIQUE** constraint is to guarantee that we can't add duplicates to our data. Furthermore, the constraint can't be applied to null values. Therefore, our affected columns can take multiple null values even if we apply the **UNIQUE** ruleset to it.

But if we don't want null values at all, we can use the **NOT NULL** constraint once again, like so:

```
CREATE TABLE BOOK_LIBRARY  
(  
  AUTHOR_NAME  CHARACTER (50),  
  BOOK_TITLE   CHARACTER (70) UNIQUE NOT NULL,  
  PUBLISHED_DATE INT  
);2
```

---

Don't forget that SQL allows us to add the **NOT NULL** constraint on columns to which the table definition refers:

```
CREATE TABLE BOOK_LIBRARY
(
  AUTHOR_NAME  CHARACTER (50) ,
  BOOK_TITLE   CHARACTER (70) NOT NULL,
  PUBLISHED_DATE INT,
  CONSTRAINT UN_AUTHOR_BOOK UNIQUE (BOOK_TITLE)
);2
```

You'll notice that in either example we have a constraint applied to the **BOOK\_TITLE** column, which means that it can't contain duplicate values or null values.



## The PRIMARY KEY Constraint

This constraint is very similar to the **UNIQUE** variant in the sense that we can also use it to ban duplicate values. Furthermore, we can use it on multiple columns, on a single column, or as a table constraint as well. The main difference between the **PRIMARY KEY** and **UNIQUE** is that if you use the first one on the column, that column can't take in a null value anymore. Therefore, we don't have to use the **NOT NULL** constrain as well, if we insert the **PRIMARY KEY**. In addition, tables can take more than a single **PRIMARY KEY** constraint.

Remember that primary keys are unique identifiers and they're important in every table. That's why we need to have the aforementioned limitations. Furthermore, in the first chapter we discussed that we can't have duplicate rows in our tables. If we have such duplicates, then whenever we change the value in one row, the same value in the duplicate row also changes. This would make the rows redundant.

What we need to do is select the primary key for our database from a pool of keys. These potential keys can be seen as groups of columns that can identify the rows in unique ways. We can make sure the potential key is unique by using either the **PRIMARY KEY** constraint or the **UNIQUE** one instead. Just make sure you add a primary key to every table, even if you lack a defined unique constraint. This way, you can guarantee you'll have unique data rows.

In order to define the primary key, first we need to specify the column (or columns) we intend to use. Here's how to apply the **PRIMARY KEY**:

```
(name of column) [ (domain) | (data type) ] PRIMARY KEY
```

We can also apply it to a table as a component by using the following syntax:

```
{ CONSTRAINT (name of constraint) }  
PRIMARY KEY < (name of column) {, (name of column) ] ... } >
```

We can also define the primary key with the use of constraints, however, we can only take advantage of this option on one column. Here's how it works:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
(  
  AUTHOR_ID INT,  
  AUTHOR_NAME CHARACTER(50) PRIMARY KEY,
```

```
PUBLISHER_ID INT);2
```

This is an example on a single column. In order to apply the key to more than one, we have to apply it to the table:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
(  
  AUTHOR_ID INT,  
  AUTHOR_NAME CHARACTER (50),  
  PUBLISHER_ID INT,  
  CONSTRAINT PK_AUTHOR_ID PRIMARY KEY ( AUTHOR_ID , AUTHOR_NAME )  
);
```

Using this method, we're introducing a primary key to the **AUTHOR\_ID** and **AUTHOR\_NAME** columns. This way we need to have unique paired values in both columns. However, this doesn't necessarily mean that we can't have duplicate values anymore inside the columns. What we're dealing with here is what the SQL pros refer to as a "superkey," which means that the primary key goes over the required number of columns.

Take note that under normal conditions we tend to use the primary key and the unique constraints on the table itself. However, to do that we need to define the constraints first. Here's how:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
(  
  AUTHOR_ID INT,  
  AUTHOR_NAME CHARACTER (50) PRIMARY KEY,  
  PUBLISHER_ID INT,  
  CONSTRAINT UN_AUTHOR_NAME UNIQUE (AUTHOR_NAME)  
);
```

Here's another example that will also work:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
(  
  AUTHOR_ID INT,  
  AUTHOR_NAME CHARACTER (50) -> UNIQUE,  
  PUBLISHER_ID INT,  
  CONSTRAINT PK_PUBLISHER_ID PRIMARY KEY (PUBLISHER_ID)  
);2
```

## The FOREIGN KEY Constraint

In the previous sections we talked about constraints that are used to ensure

the integrity of the tabular data. Remember to use **NOT NULL** to avoid null values, and the **UNIQUE** and **PRIMARY KEY** to make sure that we don't have any duplicates. Now we have another constraint to play with, namely the **FOREIGN KEY** constraint, which works differently.

Also known as a *referential constraint*, this tool is used to make a connection between the data in one table with the data in another table. The aim is to enforce data integrity in the whole database. This link between the two (or more) sets of data is what leads to the so-called referential integrity. Any changes we make to the data in one table will not have any negative effects on the information stored in other tables. Let's take a look at the examples below to get a better view of this concept.

PRODUCT_NAMES		
PRODUCT_NAME_ID: INT	PRODUCT_NAME: CHARACTER (50)	MANUFACTURER_ID: INT
1001	X Pen	91
1002	Y Eraser	92
1003	Z Notebook	93
PRODUCT_MANUFACTURERS		
MANUFACTURER_ID: INT	BUSINESS_NAME: CHARACTER (50)	
91	THE PEN MAKERS INC.	
92	THE ERASER MAKERS INC.	
93	THE NOTEBOOK MAKERS INC.	

The columns in yellow in both tables have a **PRIMARY KEY** constraint. The **MANUFACTURER\_ID** column in the first table has the same values as the column in the second table. In fact, the **MANUFACTURER\_ID** column can only take in values from the column with the same name that we have in the second table. In addition, any change we make to the first table will have some effect on the information we store in the second table. For instance, if we delete the data on one of the manufacturers, we also need to delete the information related to it from the **MANUFACTURER\_ID** column in the first table.

To do that we need to use the **FOREIGN KEY** constraint. This way we'll make sure that protected data won't be affected by any actions taken on any of the two tables. We'll achieve referential integrity on the entire database. On a related note, once we introduce a foreign key, we can refer to the table as a *referencing table*. The other table, the one that the foreign key points to, is known as the *referenced table*.

In order to set up the foreign key constraint, we need to respect some rules:

- First, we need to define the referenced table column with the use of a **UNIQUE** or **PRIMARY KEY** constraint. In most cases you should probably go with the second option.
- **FOREIGN KEY** constraints can be used on the table or a single column. We can also handle as many columns as needed when using the constraint at the table level. However, if we define it at the column's level, then we can only handle one column.
- It's important for the referencing table's **FOREIGN KEY** constraint to include all the columns we need to reference. Furthermore, the table's columns need to match the type of information of its referenced tables.
- Take note that we don't have to manually specify the reference columns. If we don't declare them, SQL will automatically see them as part of the referenced table's primary key.

If these rules don't make that much sense yet, don't worry. You'll see them soon in action. But first, let's take a look at the basic syntax of applying the **FOREIGN KEY** constraint on a column:

```
(name of column) [ (domain) | (data type) ] { NOT NULL }  
REFERENCES (name of the referenced table) { < (the referenced columns) > }  
{ MATCH [ SIMPLE | FULL | PARTIAL ] }  
{ (the referential action) }2
```

Now, let's see the syntax at the table level:

```
{ CONSTRAINT (name of constraint) }  
FOREIGN KEY < (the referencing column) { [, (the referencing column) ] ... } >  
REFERENCES (the referenced table) { < (the referenced column/s) > }  
{ MATCH [ SIMPLE | FULL | PARTIAL ] }  
{ (the referential action) }2
```

As you can see by analyzing the syntax, this type of constraint is more complicated than the other we discussed earlier. The main reason for this complexity is the fact that the syntax allows us to work with a number of options. With that being said, let's see an example of working with the **FOREIGN KEY**:

```
CREATE TABLE PRODUCT_NAMES
(
  PRODUCT_NAME_ID INT,
  PRODUCT_NAME CHARACTER (50),
  MANUFACTURER_ID INT REFERENCES PRODUCT_MANUFACTURERS
);
```

With this code we have added the constraint to the **MANUFACTURER\_ID** column. Now, in order to apply it to the entire table, we need to add **REFERENCES** and specify the name of the table. Furthermore, the foreign key's columns are identical to the primary key of the referenced table. So, if you don't need to make a reference to your target table's primary key, then you have to mention the column you want to work with. For example:

```
REFERENCES PRODUCT_MANUFACTURERS (MANUFACTURER_ID).
```

As another example, we can work with the constraint at the tabular level. Here's how to specify the name of the referenced column, though this detail isn't needed in this case:

```
CREATE TABLE PRODUCT_NAMES
(
  PRODUCT_NAME_ID INT,
  PRODUCT_NAME CHARACTER (50),
  MANUFACTURER_ID INT,
  CONSTRAINT TS_MANUFACTURER_ID FOREIGN KEY (MANUFACTURER_ID)
  REFERENCES PRODUCT_MANUFACTURERS (MANUFACTURER_ID)
);2
```

The final lines of code are essentially the definition of the constraint. As you can see, its name **TS\_MANUFACTURER\_ID** is right after the **CONSTRAINT** keyword. Take note that the name doesn't have to be specified because SQL automatically generates it if it's missing. However, it's still recommended to specify it on your own in order to avoid any possible value errors. Furthermore, automatically generated names aren't usually very descriptive, and reading your own code is much easier when you actually understand the meaning behind the names.

Now we need to declare the type of constraint we want, add the referencing

column, and then attach the constraint to it. Take note that if you have multiple columns, you need to use commas in between the names in order to keep them separate, and then use the **REFERENCES** keyword followed by the name of the referenced table. Finally, we insert the column's name as well. That's all there is to it. After we define the constraint, the **PRODUCT\_NAMES** column will no longer accept any values that aren't found in the primary key of the **PRODUCT\_MANUFACTURERS** table.

Remember that the foreign key constraint doesn't have to contain only unique values. They can be duplicated, unless you add the **UNIQUE** constraint at the column level.

With that in mind, let's add our constraint to several columns. Pay attention to the added elements to get a better idea about the syntax. We'll use the **BOOK\_AUTHORS** and **BOOK\_GENRES** tables, where the first one has its primary key defined in the Name and DOB data columns. The following SQL statement will create the second table which will have a foreign key that will reside in the **AUTHOR\_DOB** and **DATE\_OF\_BIRTH** columns. Here's how it all works:

```
CREATE TABLE BOOK_GENRES
(
  AUTHOR_NAME    CHARACTER (50) ,
  DATE_OF_BIRTH  DATE,
  GENRE_ID       INT,
  CONSTRAINT TS_BOOK_AUTHORS FOREIGN KEY ( AUTHOR_NAME, DATE_OF_BIRTH )
  REFERENCES BOOK_AUTHORS (AUTHOR_NAME, AUTHOR_DOB)
);2
```

What we have here are two referenced columns, namely **AUTHOR\_NAME** and **AUTHOR\_DOB**, and two referencing columns, which are **AUTHOR\_NAME** and **DATE\_OF\_BIRTH**. Both **AUTHOR\_NAME** columns in our tables have the same information type, and the same goes for the **DATE\_OF\_BIRTH** column and **AUTHOR\_DOB**. In this example we can conclude that we don't need matching names for our referenced and referencing columns.

## The MATCH Part

Next, we need to discuss the MATCH part of the syntax:

```
{ MATCH [ SIMPLE | FULL | PARTIAL ] }
```

Take note that the curly brackets mean we're dealing with an optional clause. Its main purpose is to give us the option to process null values within the column with a foreign key, depending on the values we might insert in the referencing column. However, if the column can't take in null values, we can't use this clause.

With that being said, this syntax provides us with three options:

1. **SIMPLE**: If we have a minimum of one referencing column with null values, we can use this option to insert any values we want to the other referencing columns. Take note that this option is automatic when we don't insert a **MATCH** section in the constraint's definition.
2. **FULL**: In order to use this option we need to have all of our referencing columns set to take in null values. If that's not the case, none of them will be able to have null values.
3. **PARTIAL**: When using this option, we may introduce null values to our referencing columns if we have a number of referencing columns that can match their referenced columns.

## The Referential Action

This is the last part of the **FOREIGN KEY** constraint syntax and it's as optional as the **MATCH** section. Its purpose is to enable us to specify the actions we want to take after manipulating the data in any of our referenced columns.

For instance, let's say we want to delete a record from the table's primary key. If we have a foreign key that references the primary key we're focusing on, our action will go against the constraint. Therefore, we need to include the information inside the referencing column in the referenced column as well. So, when we use this statement, we need to launch a certain action in the definition of the referencing table. Take note that the following action will only be triggered if the referenced table goes through any modification:

```
ON UPDATE (the referential action) { ON DELETE (the referential action) } | ON DELETE (the referential action) { ON UPDATE (the referential action) } (the referential action) ::=  
RESTRICT | SET NULL | CASCADE | NO ACTION | SET DEFAULT
```

Based on this syntax, we can either go with **ON UPDATE** or **ON DELETE**, or even both of them at the same time. Here are the actions that these clauses can

allow:

- **RESTRICT**: This is a referential action that doesn't allow us to manipulate anything that can break the constraint. The data in the referencing column can't go against the **FOREIGN KEY** constraint.
- **SET NULL**: This action allows us to set the referencing column's values to null when the referenced column is updated or deleted.
- **CASCADE**: Any modifications we introduced to a referenced column will automatically be added to the matching referencing column as well.
- **NO ACTION**: As the name suggests, this action won't allow you to perform anything that breaks the constraint. In essence, it works like the **RESTRICT** action. However, there is a difference. We can violate the information when running an SQL command, however, this violation will be stopped as soon as the command has finished its execution.
- **SET DEFAULT**: This final action allows us to update or remove the data within the referenced column and thus revert the matching referencing column to the default value. Keep in mind that this option won't do anything if the referencing column you're working with doesn't include any default value.

Now, here's an example of working with this clause. All we need to do is add it at the end of the constraint's definition:

```
CREATE TABLE AUTHORS_GENRES
(
  AUTHOR_NAME    CHARACTER (50) ,
  DATE_OF_BIRTH  DATE,
  GENRE_ID       INT,
  CONSTRAINT TS_BOOK_AUTHORS FOREIGN KEY ( AUTHOR_NAME, DATE_OF_BIRTH )
  REFERENCES BOOK_AUTHORS ON DELETE RESTRICT ON UPDATE RESTRICT
);
```

## The CHECK Constraint

This type of constraint can be used on a column, table, domain, or even within an assertion. Its purpose is to allow us to declare the values we want to introduce into our columns. Take note that we can use various conditions, such as value ranges, to define the values which our columns can contain.



This is probably the most complicated constraint even though its syntax is simple. In addition, it's very versatile. Here's how the general syntax looks:

```
(name of column) [ (domain) | (data type) ] CHECK < (the search condition) >
```

And here's the syntax for adding the constraint to a table by inserting it into the definition:

```
{ CONSTRAINT (name of constraint) } CHECK < (the search condition) >
```

Take note that we'll discuss later how this constraint can be used on domains or assertions. For now, we'll stick to the basics.

As you can see by analyzing the syntax, the constraint isn't that hard to understand. The only difficulty you can encounter here is with the search condition because it can involve complex values. The purpose of the constraint here is to test the search condition of the SQL statements that seek to change the data in the column that's protected through a **CHECK**. When the result is true, the statement will be executed and when it's false, then the statement is cancelled and we'll receive an error message.

If you're still confused about the clause, you'll probably get a better idea by practicing with a few examples. However, you should remember that the search condition's elements come with predicates; in other words, expressions that can only be used with values. In SQL these components can also be used to make a comparison between the values, for instance to check whether a value in one column is less than the value in another column.

In addition, you'll also have *utilize* subqueries as part of the condition. This type of expression works like a component that belongs to other expressions. For instance, we can use the subquery when the expression needs access to a different layer of data. Imagine having an expression that has to access the X table in order to add some data in the Y table.

It all sounds a bit complicated, so let's take a look at the following example where we use the **CHECK** clause to define the lowest and the highest values that we can insert inside a column. Take note that the constraint is generated in the table definition, and the table will have three columns.

```
CREATE TABLE BOOK_TITLES  
(  
  BOOK_ID      INT,
```

```
BOOK_TITLE      CHARACTER (50) NOT NULL,  
STOCK_AVAILABILITY  INT,  
CONSTRAINT TS_STOCK_AVAILABILITY ( STOCK_AVAILABILITY < 50 AND  
STOCK_AVAILABILITY > 1 )  
);2
```

Any values out of the 1–50 range will be rejected. Here’s another method of creating this table:

```
CREATE TABLE BOOK_TITLES  
(  
  BOOK_ID        INT,  
  BOOK_TITLE      CHARACTER (50) NOT NULL,  
  STOCK_AVAILABILITY  INT CHECK ( STOCK_AVAILABILITY < 50 AND STOCK  
  AVAILABILITY > 1 )  
);
```

Now let’s discuss the condition clause. It instructs SQL that all values inserted in the **STOCK\_AVAILABILITY** column have to be less than 50. Then we have the **AND** keyword to further instruct that a second condition needs to be respected as well. Lastly, the values have to be greater than 1.

Furthermore, the constraint enables us to list these values, and that’s why the proficient SQL users like this option when using values that don’t tend to change frequently. Now let’s define the book’s genre with the help of our **CHECK** constraint:

```
CREATE TABLE BOOK_TITLES  
(  
  BOOK_ID  INT,  
  BOOK_TITLE CHARACTER (50) ,  
  GENRE    CHAR (10) ,  
  CONSTRAINT TS_GENRE CHECK ( GENRE IN ( ' DRAMA ' , ' HORROR ' , ' SELF HELP ' , '  
  ACTION ' , ' MYSTERY ' , ' ROMANCE ' ) )  
);2
```

The values in the **GENRE** column have to be added to the items listed in the condition. Take note that we’re using an **IN** operator which forces the **GENRE** values to be added to those entries.

If you’re a bit confused by all the parentheses involved with this constraint, we can actually simplify our code by writing multiple lines instead. Let’s rewrite the above example to make it easier to read and understand:

```
CREATE TABLE BOOK_TITLES
```

```
(
  BOOK_ID    INT,
  BOOK_TITLE CHAR (50) ,
  GENRE      CHAR (10) ,
  CONSTRAINT TS_GENRE CHECK
  (
    GENRE IN
    ('DRAMA ', ' HORROR ', ' SELF HELP ', ' ACTION ', ' MYSTERY ', ' ROMANCE '
  )
)
);
```

Better? Our code certainly became more readable. In this example we need to indent each parenthesis and the data it contains because our SQL statement is built in layers. If you choose to go with this method, you'll easily notice the clauses when you return to your code. Furthermore, there is no difference between this example and the previous one. So, at the end of the day, you can go with the one you like best. Now, let's take a look at another example:

```
CREATE TABLE BOOK_TITLES
(
  BOOK_ID      INT,
  BOOK_TITLE    CHAR (50) ,
  STOCK_AVAILABILITY INT,
  CONSTRAINT TS_STOCK_AVAILABILITY CHECK ( ( STOCK_AVAILABILITY BETWEEN
1 AND 50 ) OR ( STOCK_AVAILABILITY BETWEEN 79 AND 90 ) )
);
```

Take note of the **BETWEEN** operator. Its purpose is to define the range that will include the highest and lowest values. Since we have two ranges here, we need to also keep them separate with the use of parentheses. Next, we have the **OR** keyword which is used here to form a link between those range specifications. In essence, this keyword will instruct SQL that it needs to fulfil a condition.

## Defining the Assertion

Assertions are basically just **CHECK** constraints that can be included in multiple tables. Remember that, because it means we can't create them in a table definition. With that in mind, here's the basic syntax of an assertion:

```
CREATE ASSERTION (name of constraint) CHECK (the search conditions)
```

If you compare the syntax to that of a table level **CHECK** constraint, you'll

notice that the assertion's definition is nearly identical. Once we type the **CHECK** keyword, we need to add the search conditions.

Consider the following example. We have a **BOOK\_TITLES** table with a column that contains data on how many books we have in stock. The total value, however, needs to be under the inventory we want. So, let's use the assertion to verify if the **STOCK\_AVAILABILITY** column value is less than 3,000.

```
CREATE ASSERTION LIMIT_STOCK_AVAILABILITY CHECK ( ( SELECT SUM  
(STOCK_AVAILABILITY) FROM BOOK_TITLES ) < 3000 );
```

In this example we have a subquery that is used to compare the total value to the value of 3,000. It begins with a **SELECT** keyword that is used to query the data from the table. Then we have the **SUM** function to calculate the sum of all the values in the **STOCK\_AVAILABILITY** column, followed by the **FROM** keyword which is used to set the column which contains the table. The result of our subquery is then compared to the value of 3,000. Take note that if the entry in the column is above 3,000, you'll encounter an error.

## Using the Domain Constraint

As we discussed earlier, we can also add the **CHECK** constraint to a domain definition. In this case the constraint is almost the same as all the others we discussed, with the only real difference being that a domain constraint can't be added to a specific column or table. In fact, domain constraints rely on the **VALUE** keyword in order to refer to a specified column's values. With that being said, let's create a domain using the following syntax:

```
CREATE DOMAIN (name of domain) {AS } (type of data)  
{ DEFAULT (the default value) }  
{ CONSTRAINT (name of constraint) } CHECK < (the search condition) >
```

By now you should be familiar with the components you see in this syntax. We have used the default clauses and data types before in the previous chapter. Furthermore, the constraint's definition is similar to the one definition we explored in this chapter. So, let's get straight to the example and create an **INT** domain which can only hold values ranging from 1 to 50.

```
CREATE DOMAIN BOOK_QUANTITY AS INT CONSTRAINT TS_BOOK_QUANTITY  
CHECK (VALUE BETWEEN 1 and 50 );
```

In this example we have the **VALUE** item, which refers to a specific value

inside our column through the use of the domain called **BOOK\_QUANTITY**. As before, if you add a value that doesn't match the condition of being within our mentioned range, you'll get an error.

# CHAPTER 6:

## Creating an SQL View

SQL information is stored in the database through the use of permanent tables, also known as persistent tables. However, this type of table isn't always reliable if all we need to do is check certain records. This is where SQL views come in, or *viewed tables* as some prefer to call them.

Views are essentially virtual tables. Their definitions represent schema items, with the main difference between the views and the permanent tables being that the first will not store any information. Well, to be technical about it, views don't even exist. The only thing we're dealing with is their definition because it allows us to select certain information from the table based on the statements of the definition. In order to call a view, we need to add its name to the query statement just like we do with a normal data table.

## Adding a View to the Database

Views are best used when you need to gain access to certain types of data. When using one, you might also have to define a number of complex queries and save them in the definition, because this way we can just call the view whenever we need those queries. Furthermore, using a view, you can show your data to other people without exposing anything unnecessary.

For example, you may have to present certain sections of the data on a company's employees, but without including the social security numbers or their wages. That's when the view presents itself as the ideal solution, because it allows you to present only the information you want.

## Defining the View

The simplest view you can set up is a type of view that points only to one table and gathers data from its columns without making any alterations to the information within. Here's the syntax for the most basic view:

```
CREATE VIEW (name of view) { < (name of the view's columns) > }  
AS (the query)  
{ WITH CHECK OPTION }
```

Name the view in the beginning for the definition, and name its columns as well, but only if you are in the following situations:

- When you need to obtain the values from one of the columns in order to perform the operation. Keep in mind that we're talking about a scenario where copying the values from the table isn't enough.
- When combining tables and having to deal with duplicate column names. However, you can set the column names even if it isn't necessary. For example, you might want to give your columns a more descriptive name so anyone with less experience than you can recognize the purpose of the columns.

Next, we have the second section of the syntax which includes the `AS` keyword together with a placeholder for a query. The placeholder is a basic element; however, it can involve complex statements in order to perform a set of operations. It all depends on your project and requirements. Now, let's take a look at a simple example:

```
CREATE VIEW BOOKS_IN_STOCK  
( BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY ) AS  
SELECT BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY  
FROM BOOK_INVENTORY
```

This is probably the most basic view you can set up. It calls for only three columns from the table. Keep in mind that SQL doesn't care that much about your spaces and line breaks. So, when you generate a view, you are free to list the name of the column on its own line for a better presentation of the data. A modern database management system won't complain about your style, so make sure your code is readable and consistent throughout a project.

Now let's discuss our basic example in more detail. First, we have the

declaration of the view's name, which is **BOOKS\_IN\_STOCK**. Next, we have the name of the columns and the **AS** keyword. Take note that if you don't declare the names, the view will automatically use the column names inside the table it's based on. Finally, we have the **SELECT** statement as our search expression that looks like this:

```
SELECT BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY  
FROM BOOK_INVENTORY
```

The reason we're using this basic **SELECT** statement is because it's versatile and it can be extended. We can include a number of queries in order to extract the data we need. In this case we only use two clauses, namely the **SELECT** and **FROM** clauses. The purpose of the first clause is to return the column, while the second one will prepare the table from which the data is extracted. Then we can call our view, which actually means we're invoking the **SELECT** command.

Let's discuss another example where we introduce a third clause:

```
CREATE VIEW BOOKS_IN_STOCK_80s  
( BOOK_TITLE, YEAR_PUBLISHED, STOCK_AVAILABILITY ) AS  
SELECT BOOK_TITLE, YEAR_PUBLISHED, STOCK_AVAILABILITY  
FROM BOOK_INVENTORY  
WHERE YEAR_PUBLISHED >1979 AND YEAR_PUBLISHED < 1990;
```

Take note of the last clause which sets a condition that needs to be met, otherwise the system will not extract the data for us. In this case we aren't requesting data on the authors. Instead, we use a clause to filter through the entire year's search results for every single book that was published during that time. Take note that the final clause will not have an impact on the source table. It's used to manipulate only the data that is invoked by using the view.

In addition, we can add a **WHERE** keyword to our **SELECT** statements in order to specify certain conditions and requirements. Here's an example where you can use this clause to merge the tables:

```
CREATE VIEW BOOK_PUBLISHERS  
( BOOK_TITLE, PUBLISHER_NAME ) AS  
SELECT BOOK_INVENTORY.BOOK_TITLE, TAGS.PUBLISHER_NAME  
FROM BOOK_INVENTORY, TAGS  
WHERE BOOK_INVENTORY.TAG_ID = TAGS.TAG_ID;2
```



Here we have a view called **BOOK\_PUBLISHERS**, which consists of two columns named **BOOK\_TITLE** and **PUBLISHER\_NAME**. This means we have two data sources, namely the inventory table/title column, and the tags table/name column.

But let's stick to the additional clause we introduced through the **SELECT** statement. The purpose of this clause is to qualify the columns according to the name of the table they belong to. So, if we merge the tables, we have to mention each one of them by name, otherwise there can be some confusion, especially if we have columns with duplicate names. On the other hand, if you have basic columns, then you can skip this step. With that in mind, here's how the **SELECT** clause should look:

```
SELECT BOOK_TITLE, PUBLISHER_NAME
```

Now let's move on to the **FROM** part of the statement. When we merge multiple tables, we need to specify their names by separating their entries with commas. The process is the same as we discussed above, especially when it comes to duplicate names.

Finally, we have the **WHERE** clause that will match the data rows. Take note that if we don't use this clause, we won't be able to match the values we record from various tables. So, in our example, the values from the **TAG\_ID** column, which is part of the **BOOK\_INVENTORY** table, has to match the values within the column with the same name that we have in the **TAGS** table.

Fortunately, SQL enables us to expand the **WHERE** clause. In the following example you'll see how we use this clause to restrict any returned row to the rows that have "999" in the ID column of the inventory table.

SQL allows you to qualify a query by expanding the query's **WHERE** clause. In the next example, **WHERE** restricts the returned rows to those that hold "999" in the **BOOK\_INVENTORY** table's **TAG\_ID** column:

```
CREATE VIEW BOOK_PUBLISHERS  
( BOOK_TITLE, BOOK_PUBLISHER ) AS  
SELECT BOOK_INVENTORY .BOOK_TITLE, TAGS .BOOK_PUBLISHER  
FROM BOOK_INVENTORY, TAGS  
WHERE BOOK_INVENTORY .TAG_ID = TAGS .TAG_ID  
AND BOOK_INVENTORY .TAG_ID = 999;
```

Let's see one more example using a view to obtain data from just one table.

However, in this case the view will perform some operations that will return some modified data as well.

```
CREATE VIEW BOOK_DISCOUNTS  
( BOOK_TITLE, ORIGINAL_PRICE, REDUCED_PRICE ) AS  
SELECT BOOK_TITLE, ORIGINAL_PRICE, REDUCED_PRICE * 0.8  
FROM BOOK_INVENTORY;
```

Here we have a view with three columns, namely the **BOOK\_TITLE**, **ORIGINAL\_PRICE**, and **REDUCED\_PRICE**. The **SELECT** statement is used to identify the columns that contain the data we're looking for. The title and price columns are defined through the technique we explored in the previous examples. The data inside the table's **BOOK\_TITLE** column will be copied, as well as the information from the **ORIGINAL\_PRICE** column. Afterwards, the information will be added to the columns in the **BOOK\_DISCOUNTS** view. Take note that these new columns will have the same name as the original columns. However, the last column won't be quite the same because it multiplies all the gathered values by 80% or 0.8. Those calculated values will then appear in the **REDUCED\_PRICE** column of our view.

Now let's add a **WHERE** clause to our **SELECT** statement:

```
CREATE VIEW BOOK_DISCOUNTS  
( BOOK_TITLE, ORIGINAL_PRICE, REDUCED_PRICE ) AS  
SELECT BOOK_TITLE, ORIGINAL_PRICE, REDUCED PRICE * 0.8  
FROM BOOK_INVENTORY  
WHERE STOCK_AVAILABILITY > 20;2
```

The clause is used here to limit the search only to records that have a stock availability value greater than 10. As you can see in this example, we're allowed to compare values and columns that appear in the view.

## Creating an Updatable View

Certain views can be updated by making changes to the data they contain. For instance, we can insert new rows or values. This updatability feature revolves around the **SELECT** statement. In other words, under normal circumstances, if we're using just basic **SELECT** statements, our view is very likely to be easy to update.

Don't forget that there's no particular syntax for generating an updatable view because it all depends on the **SELECT** statement following a set of rules.

The examples we had so far demonstrate how the **SELECT** statement can work as a search expression of the **CREATE VIEW** statement. Take note that query expressions can be part of other types of expressions, however, you will usually work with query specifications. In essence, a query expression is just a regular expression that begins with the **SELECT** statement and involves a number of components. In this case we can just look at the **SELECT** statement as a query specification because database elements often see it the same way.

Here are three rules you need to follow when creating this type of view:

1. The data inside this view can't be automatically merged, removed, or summarized.
2. The table must have at least one column that can be updated.
3. All columns must point to a specific column, and all rows must point to a certain row.

## How to Drop a View

If you need to remove the view from your database you can use this syntax:

```
DROP VIEW (name of the view);
```

View will be removed as soon as the **DROP** statement is processed. Keep in mind that the actual data behind the view (inside the tables) will be safe. Remember that whatever you do to a view won't cause any alterations to the data itself that was stored in a database.

Once you remove the view, it can be regenerated or you can create a new one instead.

## Database Security

One of the main reasons why we're working with fancy SQL databases nowadays is because of the security layers we can add to them. Keeping information safe is of the utmost importance, especially when working with things like personal identification or banking data. So, you need to make sure that the database can't be used or altered by anyone that lacks explicit authorization. Furthermore, you need to provide the right users with the privileges they need in order to perform their tasks.

In order to secure a database, SQL enables us to specify which information can

be seen or accessed by certain users. This is done through a security scheme, which allows us to define the actions someone can perform on the stored data. The security system works based on a set of authorization identifiers. The identifiers are just items that will represent the user who can gain access to the database and manipulate its information.

## **The Security Scheme**

As mentioned above, the security model of our database depends fully on the authorization identifiers. An identifier can be assigned to certain users in order to give them the privileges they need to access or edit database records. If the identifier doesn't define certain privileges, for instance for data manipulation, the user won't be able to change any of the information. Furthermore, identifiers can always be configured or reconfigured.

Identifiers in SQL are usually categorized as *user identifiers* or *rule identifiers*. This means that a user will have a specific profile, however, in SQL we don't have any rules set in stone on how to create a user. You can either link the identifier to the operating system itself because that's where the database system operates, or you can create the identifiers inside the database.

Roles, on the other hand, are assemblies of access privileges that can be assigned to certain users or to any other roles. So, if a role comes with access to a specific item, everyone who's been assigned to it can have the same level of access to that item. Roles are usually used to create collections of privileges that can be assigned to any identifier. One of the biggest advantages of using a role is that you can create one without having a user. Furthermore, the role will stay in the database even if we get rid of all identifiers. This makes the entire process versatile and flexible because we can always manipulate the access privileges.

The most important identifier for you to learn is the **PUBLIC** access identifier because it applies to every user. In addition, you can assign access authorizations to the **PUBLIC** profile. Just take note that when you assign these privileges to this identifier you have to make sure that the user won't use them to perform unauthorized operations.

## **Creating and Deleting a Role**

Creating a role is easy because the syntax involves an obligatory clause and an optional one. Here's how it works:

```
CREATE ROLE (name of role)
{ WITH ADMIN [ CURRENT_ROLE | CURRENT_USER ] }
```

First, we have the **CREATE ROLE** as the obligatory part of the statement, followed by the **WITH ADMIN** section which isn't absolutely necessary. In fact, this option is almost never used. The **WITH ADMIN** clause becomes viable only when the current role name and user identifier don't involve any null values. With that in mind, here's how you create a role:

```
CREATE ROLE READERS;
```

Simple as that! Once the role is defined you can assign it to any user, and if you need to delete it, all you have to do is insert the following line:

```
DROP ROLE (name of role)
```

The only important part of this syntax is specifying the name of the role you wish to remove.

## How to Assign and Revoke a Privilege

When a privilege is given, you are assigning it to an identifier. In other words, you link an identifier/privilege pair to the object.

```
GRANT [ (list of privileges) | ALL PRIVILEGES ]
ON (type of object) (name of object)
TO [ (list of authorization identifiers) | PUBLIC ] { WITH GRANT OPTION }
{ GRANTED BY [ CURRENT_ROLE | CURRENT_USER ] }
```

As you can see, here we have three must-have clauses: **ON**, **TO** and **GRANT**, followed by two optional clauses, namely **GRANTED BY** and **WITH GRANT OPTION**. Now, to revoke these privileges, all we have to do is type the following lines:

```
REVOKE { GRANT OPTION FOR } [ (list of privileges) | ALL PRIVILEGES ]
ON (type of object) name of object
FROM [ { list of authorization identifiers) | PUBLIC }
```

# CHAPTER 7:

## Database Setup

As you already know, data is stored and indexed in tables so that we can manipulate it efficiently and securely. However, before we create a table, we first need the database itself. So, if you're starting from the foundation, you need to learn how to setup and manage the database.

### Creating a Database

In order to set up the database, you need to start with the **CREATE** statement, followed by the database's name. Here's the syntax:

```
CREATE DATABASE database_name;
```

And here's an example of creating an actual database:

```
CREATE DATABASE xyzcompany;
```

Now we have an "xyzcompany" database, however, we can't use it just yet because we need to specify that it's active. To do that, we have to insert the **USE** command followed by the name of our database:

```
USE xyzcompany;
```

To access this database in the following examples, you need to use the **USE** statement once again. Now let's discuss deleting the database.

### Deleting a Database

To delete a database, use the following syntax:

```
DROPDATABASE databasename;
```

Take note that in this case, when using the **DROP** statement to delete a database, you'll have to have admin rights over the database.

## Schema Creation

Next, we need to define a schema by using the **CREATE SCHEMA** command. In addition, we can generate a number of objects and assign privileges to them in the same statement. We can also install the schema in a program. Another option would be using a number of dynamic statements instead. For instance, if you have admin rights, you can type the following statements to create a **USER1** schema and assign **USER1** to be its owner:

```
CREATE SCHEMA USER1 AUTHORIZATION USER1
```

Next, we might want to create a different schema using an inventory table and give control privileges over that table to **USER2**:

```
CREATE SCHEMA INVENTORY
CREATE TABLE ITEMS (IDNO INT(6) NOT NULL,
                     SNAME VARCHAR(40),
                     CLASS INTEGER)
GRANT ALL ON ITEMS TO USER2
```

## Specific Software Considerations

Take note that if you're working with *MySQL 5.7* you need to replace the **CREATE SCHEMA** statement with the **CREATE DATABASE** statement.

This is the syntax:

```
CREATE{DATABASE|SCHEMA}[IF NOT EXISTS]db_name [create_specification]...
create_specification:
[DEFAULT]CHARACTER SET[=]charset_name[[DEFAULT]COLLATE[=]collation_name2
```

If you're working with *Oracle 11g* instead, you can set a number of views and tables in the same operation declared through the **CREATE SCHEMA** statement. However, in order to run this statement, Oracle will process every statement inside the operation block, and if there's no error, everything will work as intended. If an error is encountered, then all the statements and changes are reset.

In other words, you can insert the **CREATE TABLE**, **CREATE VIEW** and **GRANT** statements in the **CREATE SCHEMA** command. This means that you need to have the right level of authorization to run these processes.

With all that in mind, here's how the syntax looks:

```
CREATE SCHEMA AUTHORIZATION schema_name  
  [create_table_statement]  
  [create_view_statement]  
  [grant_statement];2
```

The **CREATE SCHEMA** statement in *SQL Server 2014*, however, will generate a schema inside the database you're working with. Within this schema you can also create any tables or views. Here's how the whole process works:

```
CREATE SCHEMA schema_name_clause [ < schema_element > [, ...n ] ]  
< schema_name_clause >::=  
  { schema_name | AUTHORIZATION owner_name  
  | schema_name AUTHORIZATION owner_name }  
< schema_element >::=  
  {table_definition | view_definition | grant_statement  
  revoke_statement | deny_statement }2
```

Finally, if you use *PostgreSQL 9.3.13*, the **CREATE SCHEMA** statement will also insert a schema into the database. Keep in mind that the schema name has to be unique.

```
CREATE SCHEMA schema_name [ AUTHORIZATION user_name ] [ schema_element [ ... ] ]  
CREATE SCHEMA AUTHORIZATION user_name [ schema_element [ ... ] ]  
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION user_name ]  
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION user_name2
```

## Creating Tables and Inserting Data

As you well know, data is mainly stored in tables. To create a table, we need to give it a descriptive name, define all the columns it will contain, and determine which data types will be stored in every column.

### How to Create a Table

To create a table we'll use the **CREATE TABLE** statement, followed by an identifier and a list that will determine the columns, data type, and definition.

```
CREATE TABLE table_name  
(  
  column1 datatype[NULL| NOT NULL].  
  column1 datatype[NULL| NOT NULL].  
  ...  
);
```



This is the syntax used to create a table and it starts with the table's identifier **table\_name**. Next, we define the columns that will hold certain datatype. Each column is either defined as **NOT NULL** or **NULL**. By default, the database will consider them **NULL** if it's not specified.

But how do you determine what kind of table you need to create? Here are some of the most important questions you should be asking yourself before getting started:

- What name describes the purpose of the table with the most accuracy?
- Which data types do I need to store?
- How should I name the table columns to make them easy to understand for everyone?
- Which column do I need to designate as my main key?
- How wide should each column be?
- Which column is going to be empty and which ones will contain something?

Once you answer those questions you can create your table using the previously mentioned "xyzcompany" database. We'll call it **EMPLOYEES** because it's readable and describes the purpose of the table.

```
CREATE TABLE EMPLOYEES(  
ID      INT(6)    auto_increment, NOT NULL,  
FIRST_NAME VARCHAR(35) NOT NULL,  
LAST_NAME VARCHAR(35) NOT NULL,  
POSITION VARCHAR(35),  
SALARY   DECIMAL(9,2),  
ADDRESS  VARCHAR(50),  
PRIMARY KEY (id)  
);
```

As you can see, we now have a six-column table with the **ID** field being the primary key. The first column will contain only the integer data type and it won't take in any null values. The second column is called **FIRST\_NAME**, and as the name suggests, it's going to be a *varchar* data type column. We'll allow it a limit of 35 characters. The third and fourth columns are also *varchar* columns called **LAST\_NAME** and **POSITION**, and are also limited to 35 characters. The **SALARY** column is going to hold decimal data types with a scale of 2 and precision value of 9. Last, but not least, we have the **ADDRESS**

column which is, again, a varchar column, but with a limit of 50 characters. The primary key is the **ID** column.

## Creating a New Table Based on Existing Tables

If you use the **CREATE TABLE** statement with a **SELECT** clause you can use existing tables to create new ones. Here's how:

```
CREATE TABLE new_table AS  
(  
  SELECT [column1, column2, ... columnn]  
  FROM existing_table_name  
  [WHERE]  
);2
```

By running this code you'll generate a new table with the exact same column definitions as those in the original version of the table. Take note that you can choose any number of columns, or all of them, and copy them to the new table, adding the values from the original table to its copy. Here's an example where we create a new table called **STOCKHOLDERS** using the table above as a reference point:

```
CREATE TABLE STOCKHOLDERS AS  
SELECT ID, FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS  
FROM EMPLOYEES;
```

## How to Insert Data into a Table

In order to manipulate a database, we use SQL's aptly named Data Manipulation Language (DML). DML clauses are used to insert new information in a table or update the already existing data.

### Inserting New Data

You can add new data to a table in two ways. You either use the automatic method by using a program, or you manually enter all the information. The first method implies using an external source from which data is extracted and added to your table, or it can refer to the transfer of data from one table to another. The second method involves you and your keyboard.

Take note that data is case-sensitive and therefore you need to always double check everything you insert. For example, if you insert the name of an employee written as John, you must always type it exactly the same. JOHN is different from John or john when it comes to data.

To add data to your table you need to use the **INSERT** statement. You can use the statement in two ways. The first method is simple. All you need to do is add the information to each field and assign them to the columns of the table. This option is best used when you want to insert data into all of your columns. Here's how the syntax looks:

```
INSERT INTO table_name  
VALUES ('value1', 'value2', [NULL];
```

To use the second method, you need to insert the data in the order in which the columns appear, and you also have to add the names of those columns. Normally, this system is used when you need to insert data into certain columns. Here's how the syntax looks:

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

As you can see, in both methods we use commas to set the values and column apart from each other. Furthermore, we use quotation marks to limit the strings and datetime information.

Now, let's say we have the following information about an employee:

```
First Name: Robert  
Last Name: Page  
Position: Clerk  
Salary: 5,000.00  
Address: 282 Patterson Avenue, Illinois
```

Here's the statement we need to add all of this data to our **EMPLOYEES** table:

```
INSERT INTO EMPLOYEES (FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS)  
VALUES ('Robert', 'Page', 'Clerk', 5000.00, '282 Patterson Avenue, Illinois');
```

Next, we need to display the data we just stored in our table:

```
SELECT * FROM EMPLOYEES;
```

Take note of the asterisk (\*), which is used to inform the database management system to select all the fields in the table. Here's the output:

ID	FIRST_NAME	LAST_NAME	POSITION	SALARY	ADDRESS
1	Robert	Page	Clerk	5000.00	282 Patterson Avenue,

## Illinois

Now let's work with the next employees and add their information to the table:

First Name	Last Name	Position	Salary	Address
John	Malley	Supervisor	7,000.00	5 Lake View, New York
Kristen	Johnston	Clerk	4,500.00	25 Jump Road, Florida
Jack	Burns	Agent	5,000.00	5 Green Meadows, California

In order to add this data to our database we'll use the **INSERT INTO** statement. However, it needs to be used for each employee in every line, like this:

```
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS)
VALUES('John', 'Malley', 'Supervisor', 7000.00, '5 Lake View New York');
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS)
VALUES('Kristen', 'Johnston', 'Clerk', 4000.00, '25 Jump Road, Florida');
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS)
VALUES('Jack', 'Burns', 'Agent', 5000.00, '5 Green Meadows, California');
```

Now, let's display the updated table once again:

```
SELECT * FROM EMPLOYEES;
```

And here it is:

ID	FIRST_NAME	LAST_NAME	POSITION	SALARY	ADDRESS
1	Robert	Page	Clerk	5000.00	282 Patterson Avenue, Illinois
2	John	Malley	Supervisor	7,000.00	5 Lake View, New York
3	Kristen	Johnston	Clerk	4,500.00	25 Jump Road, Florida
4	Jack	Burns	Agent	5,000.00	5 Green

Observe how an identification number is generated for each entry. This happens thanks to the **ID** column definition where we used an attribute called “auto\_increment.” Take note that using this method, we have to specify all the columns in the **INSERT INTO** line.

## Inserting Data into Specific Columns

As mentioned earlier, you can add information to a specific column by declaring the name of the column in the column list, together with the corresponding data in the list of values. For instance, if you need to insert the name and position of an employee to a certain column, you’ll first need to specify all the column names, such as **FIRST\_NAME**, **LAST\_NAME** and **SALARY** in the list of columns, as well as the values for each of these items inside the list of values.

You already know every component of the process, so insert this data in the table.

<b>First Name</b>	<b>Last Name</b>	<b>Position</b>	<b>Salary</b>	<b>Address</b>
James	Hunt		7,500.00	

If you inserted the data correctly, your updated table should look like this:

<b>ID</b>	<b>FIRST_NAME</b>	<b>LAST_NAME</b>	<b>POSITION</b>	<b>SALARY</b>	<b>ADDRESS</b>
1	Robert	Page	Clerk	5000.00	282 Patterson Avenue, Illinois
2	John	Malley	Supervisor	7,000.00	5 Lake View, New York
3	Kristen	Johnston	Clerk	4,500.00	25 Jump Road, Florida
4	Jack	Burns	Agent	5,000.00	5 Green

Meadows,  
California

James	Hunt	NULL	7500.00	NULL
-------	------	------	---------	------

## Inserting NULL Values

At some point you'll come across a situation where you need to insert **NULL** values into your columns. For instance, when you don't have any data on the salary of a new employee, you can't just insert a random number or an estimation. With that being said, here's the syntax on inserting null values:

```
INSERT INTO schema.table_name  
VALUES ('column1', NULL, 'column3');
```

# CHAPTER 8:

## Table Manipulation

Manipulating tables and changing their structure and data is all part of the job. In this chapter we're going to discuss how to change the attributes of a column, how to alter columns, and all the rules and guidelines related to working with the **ALTER TABLE** statement. Let's start with the basic syntax used to change a table:

```
ALTER TABLE table_name [MODIFY]
[ COLUMN Column_Name] [DATATYPE | NULL NOT NULL] [RESTRICT | CASCADE]
[DROP] [ CONSTRAINT Constraint_Name]
[ADD] [COLUMN] COLUMN DEFINITION
```

We can use the first statement, namely **ALTER TABLE**, together with the **RENAME** keyword in order to edit the name of our table. For instance, we can take our **EMPLOYEES** table and change it to **INVESTORS** instead. Try it out yourself; you already have all the know-how to make this alteration.

### Altering Column Attributes

The attributes of a column are essential data properties that are added to the column. These properties are usually defined at the time when the table is initially created. However, they aren't set in stone and you can modify them with the same **ALTER TABLE** statement. After all, the name of the column is, in fact, an attribute.

You can change the following properties with this command:

- The data type of the column
- The length and scale of the column



- Whether to use null values or not

## Renaming Columns

One of the first things you'll want to change is the name of a column to make it reflect the data it contains. For example, our old **EMPLOYEES** database is now called **INVESTORS**, so the **SALARY** column from the previous database is no longer suitable. Let's rename it to **CAPITAL** and then also modify the type of data it allows to **INTEGER** with a limit of 10 digits. Remember, the old datatype for this column was **DECIMAL** and that's not the best choice for something like **CAPITAL**. Now, let's take a look at the code:

```
ALTER TABLE INVESTORS
CHANGE SALARY CAPITAL INT (10);5
```

And this is the result:

ID	FIRST_NAME	LAST_NAME	POSITION	CAPITAL	ADDRES
1	Robert	Page	Clerk	5000.00	282 Patterson Avenue, Illinois
2	John	Malley	Supervisor	7,000.00	5 Lake View, New York
3	Kristen	Johnston	Clerk	4,500.00	25 Jump Road, Florida
4	Jack	Burns	Agent	5,000.00	5 Green Meadows, California
5	James	Hunt	NULL	7500	NULL

## Deleting a Column

Due to the previous changes, one of our columns doesn't make sense anymore. So, let's delete it with the next SQL statement and then update the

table to verify the change:

```
ALTER TABLE INVESTORS  
DROP COLUMN Position; 5
```

And this is the result:

ID	FIRST_NAME	LAST_NAME	CAPITAL	ADDRESS
1	Robert	Page	5000.00	282 Patterson Avenue, Illinois
2	John	Malley	7,000.00	5 Lake View, New York
3	Kristen	Johnston	4,500.00	25 Jump Road, Florida
4	Jack	Burns	5,000.00	5 Green Meadows, California
5	James	Hunt	7500	NULL

## Adding a New Column

Now we have a new table called **INVESTORS** and we already made some new changes to it so that it can accommodate a new data set. However, we often need to insert new columns.

The process of inserting a new column is simple, and we're going to illustrate it by creating a column that will contain the total number of stocks that every investor owns. Let's call this column **STOCKS** and make it hold only integers with a limit of nine digits. Here's the statement, followed by an updated version of the table.

```
ALTER TABLE INVESTORS ADD STOCKS INT(9);5
```

ID	FIRST_NAME	LAST_NAME	CAPITAL	ADDRESS	STOCKS
1	Robert	Page	5000.00	282 Patterson Avenue, Illinois	NULL
2	John	Malley	7,000.00	5 Lake	NULL

				View, New York	
3	Kristen	Johnston	4,500.00	25 Jump Road, Florida	NULL
4	Jack	Burns	5,000.00	5 Green Meadows, California	NULL
5	James	Hunt	7500	NULL	NULL

## Alter a Column without Modifying the Name

In SQL it's possible to change a table in numerous ways, while also leaving the initial name intact. For instance, we can use the **MODIFY** command together with the **ALTER TABLE** statement. In the following examples, we'll use this combination of commands to alter the **CAPITAL**'s data type from integer to decimal, and limit it to nine digits and two decimals. Here's the full statement:

```
ALTER TABLE INVESTORS MODIFY CAPITAL DECIMAL (9,2) NOT NULL;
```

Now you can double check the name of the columns and their properties to see what changed. All you need is the **SHOW COLUMNS** command and you'll see all the columns our table contains. Here's an example:

```
SHOW COLUMNS FROM INVESTORS;
```

Field	Type	Null	Key	Default	Extra
ID	int(6)	NO		0	
FIRST_NAME	varchar(35)	NO		NULL	
LAST_NAME	varchar(35)	NO		NULL	
CAPITAL	decimal(9,2)	NO		NULL	
ADDRESS	varchar(35)	YES		NULL	
STOCKS	int(9)	YES		NULL	

## Using ALTER TABLE and a Few Rules

There are a couple of general rules and guidelines when manipulating your data with the **ALTER TABLE** command. The first thing to keep in mind is that when you insert a new column, you can't have the **NOT NULL** property attached to it when you're adding it to a table that already contains data. This property should be specified only to show that it will store a value. A **NOT NULL** column can counter the constraint if the table's data doesn't contain any values for the new column.

When you make changes to the columns and the fields you need to take note of the following aspects:

- Changing the column's datatype is an easy process and you shouldn't be discouraged from doing it.
- When you alter the number of digits that a value is allowed to contain, you need to pay attention to the limit of digits that can be stored by a table.
- Decimal places can also be altered easily, but again there's a limit, and you need to check and see that you aren't going overboard with the number of digits that can be stored.

Above all, you need to verify the changes you make and check the updated information and make sure the new attributes won't cause data losses, or any other trouble when executing an **ALTER TABLE** command.

Furthermore, when you drop a table, you need to remember that all of the data, constraints, and everything related to it will be removed. For instance, if you delete the **INVESTORS** table and then you try to use the **SELECT** statement on it, you'll get an error telling you that the data doesn't exist.

# CHAPTER 9:

## Time

Dealing with time-specific problems in programming as well as database management takes a bit of know-how. In the old days, for instance, SQL couldn't tell the difference between data that was valid only in a certain timeframe and then later was invalidated by new data. Therefore, developers had to focus a lot of their time on maintaining accurate data through manual updates, instead of automating the process through a flexible database and data management system. Fortunately, modern data models and today's SQL are designed with temporal data in mind, therefore it can process and manage it without always requiring direct attention.

In this chapter we are going to focus on understanding temporal data, which is any form of data that's linked to a specific timeframe during which it was valid. We will learn how to figure out whether your information is still relevant and you will discover that there are different temporal data types that can have an impact on the elements of the database.

## Datetime Data Types

Real world data will often involve time values, so you need to familiarize yourself with SQL's five datetime data types so that you can work with these values. Take note that a number of features found in one type will also be found when working with a different type because they often intersect. In addition, some of them may only be valid on certain SQL implementations, meaning that if you transfer a database from your current model to a different one, you may have to fix and modify a few things. You should think of this

even when dealing with other data types and formats. The best approach is to check the documentation of the implementation, make a backup of your data, and only then attempt to move it.

With that in mind, let's briefly go through the six datetime data types you need to know:

1. **DATE:** The first type is the most basic one and you'll work with it frequently. The **DATE** datetime will contain three data objects, namely the year, month, and day. Furthermore, there are automatically placed restrictions on the number of digits you can use for each object. For instance, the month and day data items can only have two-digit values, while the year can have four digits. There's also a restriction on the year range because the system will allow you to add data starting from year one (presented as 0001 due to the 4-digit restriction) all the way to year 9999. Finally, you need be aware that the year, month, and day objects are written with dashes in between and therefore the length of the value must have 10 spaces. It is written in this format: 2020-05-31.
2. **TIME WITHOUT TIME ZONE:** As the command for the data type suggests, this is used to store information that contains the hour, minute, and second time objects. In some ways it's similar to the **DATE** data type because the temporal items are also restricted to a number of digits. For instance, hours and minutes can only be written with two digits, while seconds can have at least two, but more are possible. Here's how the format is stored: 12:55:11.676. Take note that we also have decimals when storing seconds and that the time objects are separated by colons. Furthermore, the data type can be declared using the **TIME** keyword alone, however, this won't include the use of decimals. Therefore, if your time stamp needs to be accurate to the millisecond, then you need to use the **TIME WITHOUT TIME ZONE (y)** command, where "y" is a fill in for the number of fractional digits you want to allow.
3. **TIME WITH TIME ZONE:** This SQL datetime is similar to the previous one, however, it also enables us to store information on the time zone, the default being Universal Time (UTC). The value of the this datetime object can be anything from -12:59 to +13:00

and it must be written in this format that takes six spaces. The time zone information is also added only after the time data by separating them from each other using a hyphen. The plus and minus signs are also mandatory because they are used to declare the temporal offset. Finally, we can use decimals just like in the **TIME WITHOUT TIME ZONE** example.

4. **TIMESTAMP WITHOUT TIME ZONE:** The timestamp data type is used to store the date as well as the time. This data type still comes with a number of constraints, but they are the same as in the previous examples. Just look at the **DATE** restrictions for the date object and at the **TIME** restrictions for the time object. In addition to having the date and time in a single value, however, we also have default fractional values for the temporal data. In the other examples we need to specify when we want to use decimals, and here we have a default value set to 0, which we can modify up to 6 digits.
5. **TIMESTAMP WITH TIME ZONE:** As you can probably guess, the only real difference between this data type and the previous one is that we can store the time zone object next to our temporal data. If you need every possible detail about time, you should use this data type.
6. **INTERVAL:** Yes, there are five datetime data types, however, the interval data type is related to the others, even though technically it doesn't belong in the same category. The purpose of this data type is to represent the difference between two distinct points in time, whether that time is formulated as dates or day time. Therefore, it can be used in two scenarios: when we calculate a period between two dates and when we calculate the period between two different parts of the same day.

These are the basic datetime data types you'll be working with in SQL. However, there's much more to them, and in the following section we are going to discuss more about the period data type, which is more complex than the **INTERVAL** type.

## Time Periods

As mentioned in the previous section, a period can be defined between two

dates, or two time positions. The problem is that SQL doesn't have a well-defined period data type because the idea of the period came very late to the table and the SQL developers didn't want to risk damaging the well-defined structure of the language. Therefore, period definitions are found as metadata. This means they are part of the data tables as an actual element because there's a column that stands for the start of the period and a column to mark when the period ends. To control this aspect of our tables we have to modify the creation, and alter table syntax.

Statements can be used to add the functionality that enables us to either declare or delete a time period. Because the period is represented by normal columns, the same rules apply to them as for any other columns. SQL is simply used to label the starting point of the period, while the end is controlled by the database management system through a constraint. The constraint is used to ensure that the end of the period won't be smaller than the defined start marker.

Take note that there are two temporal dimensions that we need in order to work with this kind of data. They are referred to as the transaction time and the valid time. The first represents the period when a specific piece of information was inserted in our database. The second dimension describes the period during which our data is valid, meaning it shows the present reality.

The two dimensions can have different values because our data can be stored in a database before it becomes valid. Let's say we're working with some data based on a contract between two parties. That contract can be agreed upon months before any of the data in it becomes valid. We can store the information, but it's not yet reflecting the reality as the contract hasn't come into force yet.

Finally, we can also set up tables to represent each dimension, or we can define them both in the same table. In this case, you need to be aware that the transaction time dimension is stored in system-controlled tables, and therefore the time frame is attached to the system time. On the other hand, the valid time only depends on the time presented by the program that relies on it.



## Time Period Tables

The best way to understand the concept we discussed earlier is by jumping straight in the middle of an example. So, imagine that a company wants to store some information on its employees during a period of time by dividing them into categories that match their departments. Here's how this kind of table would look:

```
CREATE TABLE employees (  
  EmployeesID INTEGER,  
  EmployeesBeginning DATE,  
  EmployeesEnding DATE,  
  EmployeesDepartment VARCHAR (25) ,  
  PERIOD FOR EmployeesPeriod (EmployeesBeginning, EmployeesEnding) ) ;
```

Now that we have the table, we need to add some temporal data:

```
INSERT INTO employees  
VALUES (4242, DATE '2018-02-02', DATE '3000-12-31', 'TechAssistance' );
```

As you can probably conclude from the code itself, our end date (year 3,000) is used here to say that the employees' data will remain valid for a long time, or at least as long as they'll still be working at this company. Take note that we can also specify the exact time limit if we want, but in this case, we just went with a period that can't be too short.

Now start asking yourself what will happen when one of the employees is moved from one department in 2019 and then in 2020, he is assigned back to his previous department. We need to change our database to take this action into account.

```
UPDATE employees  
FOR PORTION OF Employeesperiod  
FROM DATE '2019-12-03'  
TO DATE '2020-06-03'  
SET EmployeesDepartment = 'Development'  
WHERE EmployeesID = 42;
```

After the update, we now have three new rows. In the first one we have information on the employment period before the departmental reassignment happened. The second one contains the temporal data on the time of employment spent in the new department. Finally, the last row begins a new period that marks that employee's return to his initial department.

Next, we're going to delete some of the temporal data as the removal process is a bit different from deleting data in a regular table. We can't just start deleting the rows. Imagine if the employee in our example didn't move to another department, but instead he quits the company and at a later date he returns. This is a situation (a bit unlikely in this particular example) that requires us to delete only part of the temporal information, so here's how our delete statement would look:

```
DELETE employees
FOR PORTION OF EmployeesPeriod
FROM DATE '2018-12-03'
TO DATE '2020-06-03'
WHERE EmployeeID = 42;
```

The period in the middle that we had in the previous example no longer exists. Now we have only the first-time frame containing data on the employee's initial hiring, and the last one which represents the return to his original department.

Now, did you notice anything weird about our tables? There's no primary key. All we have is an employee ID, but it's enough for us to use it as an identifier in our example. However, the table can contain multiple rows of data for every single employee and thus the ID is not enough to work in place of a proper primary key. Using our example, we can't make sure we always have unique values, and that's why we should insert the `EmployeesBeginning` and `EmployeesEnding` into the key. Unfortunately, this won't fix all of the potential problems.

Look at our table where one of the employees moved between departments for certain periods of time.

If the starting and ending points of a time frame are added to the primary key, then we'll have unique objects. The problem here, though, is that these periods can overlap and therefore some employees might show up as working for both departments at the same time for a certain amount of time. This is what we would refer to as corrupted data. So, we need to get this fixed. The easiest approach would be a constraint definition that tells the system that the company's employees can only work for a single department. Here's an example by using the `ALTER TABLE` statement:

```
ALTER TABLE employees  
ADD PRIMARY KEY (EmployeesID, EmployeesPeriod WITHOUT OVERLAPS) ;
```

We can also introduce the constraint when we set up the data table:

```
CREATE TABLE employees (  
EmployeesID      INTEGER NOT NULL,  
EmployeesBeginning DATE  NOT NULL,  
EmployeesEnding  DATE  NOT NULL,  
EmployeesDepartment VARCHAR (25) ,  
PERIOD FOR EmployeesPeriod (EmployeesBeginning, EmployeesEnding)  
PRIMARY KEY (EmployeesID, EmployeesPeriod WITHOUT OVERLAPS) );
```

With this final form of our table, we no longer run the risk of intersecting rows and corrupted data. In addition, we introduced the **NOT NULL** constraints just in case we have a database management system that doesn't deal with possible null values on its own. Not all systems handle them automatically for us, so it's better to plug any potential leaks from the start.

## System Versioned Tables

Another type of temporal table is the system version table that serves a different purpose. Remember that the tables in the previous section work based on particular periods of time and only the data that's valid during that period is going to be processed and used.

The purpose of system versioned tables is to enable us to determine the auditable data on objects that were either newly inserted, altered, or deleted from the database. For example, imagine a bank that has to determine the time when a deposit was made. This is information that has to be immediately registered and then managed for a specific amount of time depending on laws and regulations. Stock brokers operate the same way because an accurate temporal record can have a serious impact on other data. System versioned tables make sure that all this information is accurate to milliseconds. With this example in mind, here are two conditions a program needs from this type of data table:

1. The table rows have to exist in their original form even after being altered or deleted. In other words, the original data has to be preserved.
2. Each time frame has to be processed by the system and the

program needs to be aware of this aspect.

The first rows that we alter or remove will remain part of the table, however, they'll appear as historical rows and we won't be able to change them. In a way they're backed up information in its original form. Furthermore, we can't make future alterations to the time frames attached to this data. But this doesn't mean that the time periods can't be changed. It only means that we as users can't alter them any further, but the system can. This aspect of system versioned information is important because it acts as an additional security measure by not allowing any user to make modifications or remove the original data and the time frames during which it was valid. This way audits can be performed on the data and government agencies can always have access to it because tampering is nearly impossible.

Now that you know more about both types of tables, let's compare them to get a better understanding of the difference between a period table and a system versioned table:

- When working with period tables, we can define the period and the name of the time frame ourselves. On the other side, when it comes to system versioned tables, the period is defined under **SYSTEM\_TIME**.
- Another big difference is related to the **CREATE** command. To have a system versioned table, we need to introduce the **WITH SYSTEM VERSIONING** statement. In addition, we should use the timestamp datetime data type when we declare the time frame starting and ending points. This, however, is just a recommendation because we should look for the best accuracy we can get out of the system. As previously discussed, the timestamp data type is the most precise.

Let's now see an actual system versioned table and how it's created:

```
CREATE TABLE employees_sys (  
  EmployeesID      IINTEGER,  
  SystemStartingPoint  TIMESTAMP (12) GENERATED ALWAYS AS A ROW START,  
  SystemEndPoint      TIMESTAMP (12) GENERATED ALWAYS AS A ROW END,  
  EmployeesName      VARCHAR (40),  
  PERIOD FOR SYSTEM_TIME (SystemStartingPoint, SystemEndPoint))  
WITH SYSTEM VERSIONING;
```

Notice that we identify the valid system row by checking whether the time frame is within the system time. If it's not, then we have a historical row instead.

The two systems we've been working with are different, as you've seen. But now that you're familiar with the syntax, you may have spotted other different elements such as:

1. The system start and system end points are automatically generated by the database management system. We always add the "generated always" statement as a result of this.
2. In system versioned tables the **UPDATE** and **DELETE** statements will work on the valid system rows alone, and not on the historical rows. Historical rows can't be altered and this includes the starting and ending points of the period.
3. Another difference is that when you use the **INSERT** command to add a new component to the table, the system start column is, by default, linked with the transaction timestamp. Remember that this element is connected to all data transactions.
4. Finally, when we delete or update any of the data in the table, a historical row containing that information will be created automatically.

Whenever you update a system versioned table, the original row with its time periods will be added to a transaction timestamp to register that the data row is now invalid and no longer processed. In addition, the database management system you use will update the beginning of the period based on that timestamp. So, the row we just updated is now the active row that's valid and can be used in any SQL operations.

As a final note on this topic, you should remember that assigning a primary key to a system versioned table is a simple process, and is in fact even more basic than working with the time period tables. This is all due to not having to deal with time periods in particular. Remember that historical data rows cannot be altered or deleted, but we already know that valid rows contain unique values because they're automatically checked by the system. So, when the row can't be changed, it can no longer be checked either.

With all that in mind, let's assign a primary key to our table with the **ALTER** statement. Here's the syntax:

```
ALTER TABLE employees_sys  
ADD PRIMARY KEY (EmployeesID);
```

# CHAPTER 10:

## Database Administration

If your goal is to become a data analyst or database admin, you need to learn how to administer a database, how to back up all the data, and how to restore it if necessary. These are the foundational elements of database maintenance and we are going to go over them in this chapter.

### Recovery Models

SQL comes with a number of recovery models that you can use on a database. What's a recovery model, you ask? It's simply an option that allows the database to determine which data types can be backed up or restored. For instance, if you're working with valuable information, or you need to copy the production environment, you'll need a recovery model to have your back. Just take note that the selected model will need some time to back up or restore an entire database. So, if you're dealing with a massive amount of data, you need to factor in that time.

In this section we are going to discuss a number of recover models, followed by an SQL statement to show you how to set them up and use them in a real scenario.

The first recovery model is the full model which will back up everything in your database and keep track of your history. The model works by saving a number of "point in time" backups, which means you can revert the database to any of these points. So, if you lose some data, you can restore one of these points and recover it. If all of your data is important, you should probably go

with this option because it saves everything.

Here's the syntax for this recovery model:

```
ALTER DATABASE DatabaseName SET RECOVERY FULL
```

Next up we have the simple recovery model that also backs up all the information inside a database, however, the history log is emptied. This means that when we restore the database, the log will be recycled, and from this point on it will record only the new activities. Take note that the log can't be rolled back to recover data if you delete it or lose it somehow by accident. This model is best used when working with a new server and you have to immediately restore a database, or copy one in order to use it as a testing environment. With that being said, here's the syntax for the simple model:

```
ALTER DATABASE DatabaseName SET RECOVERY SIMPLE
```

Another recovery model you'll encounter is the bulk logged model. As the name suggests, it offers us a fast backup/restore process because it doesn't store the bulk operations in the transaction log. This makes the whole recovery process much faster. The model itself works just like the full model because it allows us to use various points in time to recover lost data.

So, if you're handling crucial information and you're using a lot of bulk statements in your database, you should probably use this model. However, if you don't have many bulk statements, the full recover model should be used instead. Here's how the syntax looks for this recovery model:

```
ALTER DATABASE DatabaseName SET RECOVERY BULK_LOGGED
```

Now, let's assume that we're working with a huge database and we need to back it up for recovery. You can use any of the syntax models above, depending on your situation. The easiest way would be to set your **Company\_Db** model to full, and then execute the backing process.

## Database Backup Methods

There are several methods to back up a database, and each one of them depends on which recovery model you chose. So, let's discuss all of our options.

If you used the *full recovery* method, you can back up the entire database



together with its log. The full backup option is best used when you need to add a database to another server so that you can start testing it. Take note that as it was with the recovery process, if the database is large, the backup process will take some time. However, when you're backing up the entire database, you don't have to repeat the operation too often, certainly not as much as when using the differential method.

Here's how the syntax for the full backup method looks:

```
BACKUP DATABASE DatabaseName TO DISK = 'C:\SQLBackups\DatabaseName.BAK'
```

Next, we have the *differential backup* method which depends on already having a full backup. This means you need to perform the full backup first. The purpose of this method is to back up only new changes that occurred between the last full backup you performed and the differential backup process you're running now. This method can be used in a variety of scenarios, however, due to the way it works you need to execute it more often since you're introducing or updating new data to the database. Luckily this is a fast process and it doesn't require much data storage space.

Here's how the syntax for the differential backup method looks:

```
BACKUP DATABASE DatabaseName TO DISK = 'C:\SQLBackups\DatabaseName.BAK'  
WITH DIFFERENTIAL
```

Finally, we have the *transaction log*. Take note that the log depends on having a full backup and it will back up any new activities that we performed in our database. The transaction log is very useful, especially when you need to perform a full database restoration because it records all the activities in the history of the database. So, you should run this process as often as you can.

Having a frequently backed up transaction log means that your log won't take up much space because they become smaller and smaller as you back them up. However, this method depends on the type of database activities in your project. Therefore, if you have a database that isn't highly active and you back it up on a regular basis, the process will be fast and restoring the data will be painless as well.

Here's how the syntax for backing up the transaction log looks:

```
BACKUP LOG DatabaseName TO DISK = 'C:\SQLBackups\DatabaseName.TRN'
```

Now, take some time to practice with an exercise. Define the **Company\_Db** database's recovery model to full and back it up using the full backup technique. Afterwards, start removing the data from the **Sales.Product\_Sales** table. Take note that the purpose of this part of the exercises is to simulate a loss of data. Afterwards, query the table which lost all that information just to verify whether the table still exists. Finally, you can restore the database with all the information you removed from the sales table.

## Restoring a Database

The most important part of restoring a database is first having a full-proof backup. Only then can you choose the right data recovery model.

The first thing you must observe is how the backup file contains a number of files and file group inside it. Every file can either be a differential backup or a full backup, therefore you need to learn which one you should restore. In addition, you need to specify the files that should be restored. If you don't, then SQL will choose the default file, which normally is a full backup file.

So, start by first learning what files you have inside the **BAK** file by using the **RESTORE HEADERONLY** statement. Always take this step before committing to a database restoration. Then you can type the following line to view the backup files:

```
RESTORE HEADERONLY FROM DISK = 'C:\SQLBackups\DatabaseName.BAK'
```

Now, look at the number in the column titled "BackupType." The value of 1 represents the full backup, while 5 refers to the differential backup. In addition, you need to use the number in the position column to identify the file you're looking for. In the following example, we'll be working with file number 8, which you can see is a full backup, and file number 9, which is a differential backup.

	<b>BackupName</b>	<b>BackupDescription</b>	<b>BackupType</b>	<b>ExpirationDate</b>	<b>C</b>
1	NULL	NULL	1	NULL	0
2	NULL	NULL	1	NULL	0
3	NULL	NULL	1	NULL	0
4	NULL	NULL	1	NULL	0

5	NULL	NULL	1	NULL	0
6	NULL	NULL	1	NULL	0
7	NULL	NULL	5	NULL	0
8	NULL	NULL	1	NULL	0
9	NULL	NULL	5	NULL	0

Furthermore, you need to work with the master database and then declare the wanted database as a non-active database, meaning **SINGLE\_USER**, so that you enable just one connection before the restoring process. Take a look at the following syntax, where we use **WITH ROLLBACK IMMEDIATE** in order to roll back any unfinished transaction logs in order to set our database to a one-connection state.

In this example we're setting the database to enable a single open connection, and the rest will be closed.

```
ALTER DATABASE DatabaseName SET SINGLE_USER WITH ROLLBACK
IMMEDIATE
```

When the restoration process is over, we can reset the database to its active state, namely **MULTI\_USER**. Look at the syntax below where we enable multiple users to connect to the database.

```
ALTER DATABASE DatabaseName SET MULTI_USER
```

## Restore Types

In this section we are going to discuss several database restore types you can use based on your situation.

The first is the *full restore* type, which is used to restore the whole database and all of its files. Take note that you can also use the **WITH REPLACE** clause to overwrite an already existing database. However, if you set the recover model to Full, you'll need to use this clause anyway. Furthermore, if you use the **FILE = parameter** action, you can select the file you want to restore, whether you're performing a full or differential restore. On the other hand, if you went with the simple recovery model, you can skip using the **WITH REPLACE** clause. Finally, this action will create the database, its files, and all of the information within if it doesn't already exist, because it's restoring

everything from the **.BAK** file.

Here's an example of restoring a file when using the full recovery model:

```
RESTORE DATABASE DatabaseName FROM DISK = 'C:\SQLBackups\DatabaseName.BAK'  
WITH FILE = 1, REPLACE
```

And here we have an example when using the simple recover model instead:

```
RESTORE DATABASE DatabaseName FROM DISK = 'C:\SQLBackups\DatabaseName.BAK'
```

Next up we have the differential restore option. Again, you should start by using the **RESTORE HEADERONLY** statement to learn which backup files are available and whether they're full or differential file types. Then you can choose to run a full restore of the backup file using the **NORECOVERY** option, because this specifies that other files should be restored as well. When the backup is specified, you can also use the **WITH NORECOVERY** option in order to restore the differential file types, too. Lastly, if you add the **RECOVERY** statement in the differential file type restore, you'll be telling the system that there are no other files that require restoring.

Here's the syntax for restoring the full backup file:

```
RESTORE DATABASE DatabaseName FROM DISK =  
'C:\SQLBackups\DatabaseName.BAK' WITH FILE = 8, NORECOVERY
```

Here's the syntax for restoring the differential backup file:

```
RESTORE DATABASE DatabaseName FROM DISK =  
'C:\SQLBackups\DatabaseName.BAK' WITH FILE = 9, RECOVERY
```

And finally, to run a perfect database restore process you need to use the log restore option. Once you perform the full or differential restore of the database, you can restore the log as well. Take note that you also need to use the **NORECOVERY** statement when you restore the backup so that you can restore the log.

Now, going back to the backup you made for your database and the data loss simulation performed on the **Product\_Sales** table, you can now restore the database. To do that, you need to use the master database. Set the database to a one user state, run restore process, and then set it to the multiuser state. After the database has been restored, you can recover the data from the table just to confirm that you have access to that information once more.

## Attaching and Detaching Databases

The techniques used to attach and detach databases have many similarities with the methods used to back up or restore a database. Essentially, you'll be copying **MDF** and **LDF** files, and performing a process similar to back up and restore, but faster. Furthermore, the database has to go, which means that nobody can have access to it in this state. It will remain this way until we reattach it.

Since the process of backing up/restoring and attaching/detaching are similar, we need to determine when to use which one. Normally, you should go with the backup solution, however, you'll encounter scenarios when all you can do is go with the second option.

Imagine having a database with a large number of file groups. Attaching it can be difficult and time-consuming. So, in this case you should back it up and restore it to a different destination. During the backup process, the files will group together automatically.

Now, if you have a huge database, the backup process might take a very long time. This is when we can take advantage of the speed offered by the attaching/detaching method. It works quickly because the database is taken offline, detached, and then simply attached to a different location.

The two file groups that can be used with this method are the **.MDF** and **.LDF** files. As you probably noticed, the **.MDF** file is actually the main database file that contains all the data and information on the structure of the database. The **LDF** file, on the other hand, deals with only the history of the transactional log and activities. Then we have the **BAK** backup file, which automatically groups all the files, and can be used to restore various versions of those files.

Choose the right method after analyzing the scenario you're in. However, the backup/restore method should still be your preferred option in most cases. Just examine your situation and perform your tests before working with live data.

We've attached the database, so let's detach it from the server and then perform the attaching process once more with the SQL syntax.

## **Detaching the Database**

Take note that in *SQL Server* we have a stored method of detaching a database,

and it can be found inside the master database. Here's how you can peak inside the procedure and analyze its complexity:

1. Expand the database folder.
2. Select System Databases, followed by the Master database.
3. Select Programmability.
4. Click on Stored Procedures, followed by System Stored Procedures.
5. Locate sys.sp\_detach\_db, click on it with the right mouse button, choose Modify, and then you'll have access to the syntax.

Start executing the procedure as you see it there. Here's how the syntax looks:

```
USE master
GO
ALTER DATABASE DatabaseName SET SINGLE_USER WITH ROLLBACK IMMEDIATE
GO
EXEC master.dbo.sp_detach_db @dbname = N'DatabaseName', @skipchecks = 'false'
GO
```

To gain a better understanding of what's going on in this syntax, you need to use the master database in order to change the database that's about to be detached. It also has to be declared as a single user database. In addition, after **@dbname** we have a value that enables you to declare the name of the database that you want to detach. Then the **@skipchecks** is set to false, which means that the database system will update the data and confirm that the specified database was detached. This part should be set to **@false** when the database you're detaching contains any data on other databases.

## Attaching Databases

Once the database has been detached, you can navigate in the data directory and check whether you still have the database's **MDF** file. Naturally, it should be there because we haven't deleted anything, just performed a detaching process.

Now, copy the **MDF** file's path and paste it somewhere where you can have it readily available, for instance in a Notepad file:

```
The location is C:\Program Files\Microsoft SQL
Server\MSSQL13.MSSQLSERVER\MSSQL\DATA.
```

When you connect to the instance and launch a query session, you need to specify where the data should be stored, and that's what this path is for. Afterwards, you can type the value in the following syntax so that you

reattach the database. Take note that in the following example we have a syntax that will reattach both database and log files. Therefore, you don't have to attach the log file entirely because we aren't attaching it to a server.

```
CREATE DATABASE DatabaseName ON (FILENAME = 'C:\SQL Data  
Files\DatabaseName.mdf'), (FILENAME = 'C:\SQL Data Files\DatabaseName_log.ldf') FOR  
ATTACH
```

We start the example by first using the attach statement to attach the log file if we have one. Take note that if there's no **LDF** file and only an **MDF** file, you won't encounter any issues. The main data file will be attached and the database system will simply generate a fresh log file and record the activities to it.

Now, let's say we need to detach our database from a server so that we can reattach it to a new one. You can use the syntax we just discussed in order to perform this process using an existing database, such as the **AdventureWorks2012** database. You can find this sample database on Microsoft's website by following this link: <https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms>.

For this exercise, imagine that you have to detach this database from an old server and that you're going to attach it to a new server. Go ahead and use the syntax above as an example in order to attach the **AdventureWorks2012** database. Here's an example from the **Production.Product** table you can work with:

<b>ProductID</b>	<b>Name</b>	<b>ProductNumber</b>	<b>Color</b>	<b>StandardCost</b>	<b>ListPrice</b>
317	LL Crankarm	CA-5965	Black	0	0
318	ML Crankarm	CA-6738	Black	0	0
19	HL Crankarm	CA-7457	Black	0	0
320	Chainring Bolts	CB-2903	Silver	0	0
321	Chainring	CN-6137	Silver	0	0

## Nuts

---



# CHAPTER 11:

## Logins, Users and Roles

### Server Logins

A server login is essentially an account that's generated on an SQL server and it relies on the server authentication. In other words, the user has to log into the server instance using a unique username and password.

A server login will allow you to log into the SQL Server, however, this doesn't mean you'll automatically have access to the database. That level of access depends on the assigned privileges. Furthermore, access can also be assigned to different server roles that include only certain privileges and permissions.

### Server Roles

The purpose of the server roles is to determine which permissions and privileges a user will have. Here are the preset roles at the server level:

- **Sysadmin:** With this role active you can do anything on the server. A system admin holds all possible privileges.
- **Server admin:** This role can manipulate the configurations of a server or even take it offline.
- **Security admin:** This role revolves around both server and database level privileges, mainly focusing on granting, denying, and revoking access permissions. Furthermore, the security admin is in charge of the password administration and can reset them as necessary.

- **Setup admin:** This role enables you to add or delete linked servers.
- **Disk admin:** This role gives you access to the commands used to manage disk files.
- **Process admin:** This role comes with the power to stop any process from running on the server.
- **Database creator:** This role has control over the **DDL** statements, meaning that it comes with the power to create, modify, remove, or restore a database.
- **Public:** This role has default logins with no special permissions, and privileges fall into the public role.

If you want to learn the details of all of these roles you can [follow this link](#) and go through Microsoft's documentation on the subject.

Now, let's see the syntax that we'll use to set up a server login. Take note that in the following example you can switch the **dbcreator** role with any other role you want and then assign it to the user login. Furthermore, take note of the square brackets we're going to use when declaring the master and user names because they're used as identifiers in this case.

Let's start by using the master database.

```
USE [master]
GO
```

The next step is to create the login. In our example we'll have "User A" as the user name, but you can use whatever you want. Then we'll have a password field where we'll add the new password. Finally, we'll make the master the default database.

```
CREATE LOGIN [User A] WITH PASSWORD= N'123456',
DEFAULT_DATABASE=[master]
GO7
```

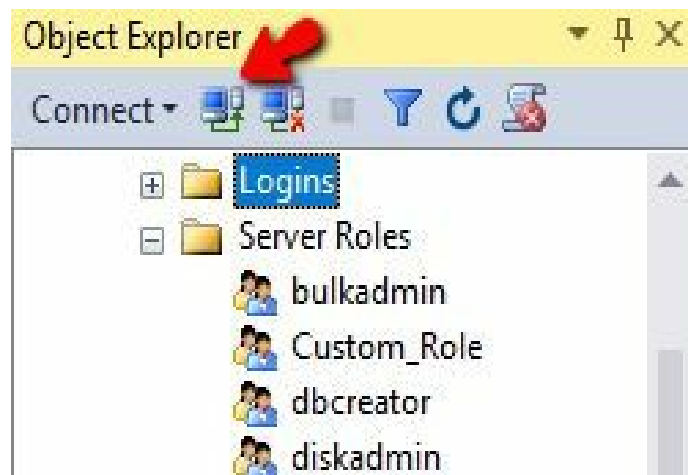
## Assigning Server Roles

In order to assign the server level roles, we can use several methods. One of the more popular ones involves using the *Management Studio* interface, and another one is using pure syntax. Let's first start with the syntax. In the following example we are going to give the user the role of **Dbcreator**. Remember that in the previous example we only created the login for our user, we didn't assign anything.

```
ALTER SERVER ROLE dbcreator ADD MEMBER [User A]
GO
```

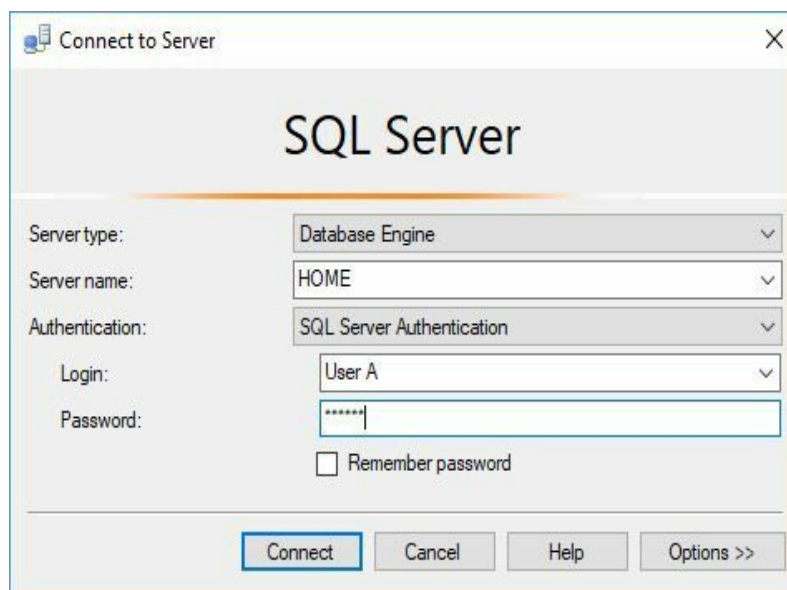
Now that we have a server user, change their role to that of a server admin. Just follow all the instructions you went through in the previous section. Once you have accomplished this small objective, you should see a message that tells you the operation was successful and that you can log into the SQL Server with the credentials attached to that user account.

Now, let's look at the *Management Studio* option. In order to log in, head to the Connect Object Explorer button and click on it to see the following connection window.

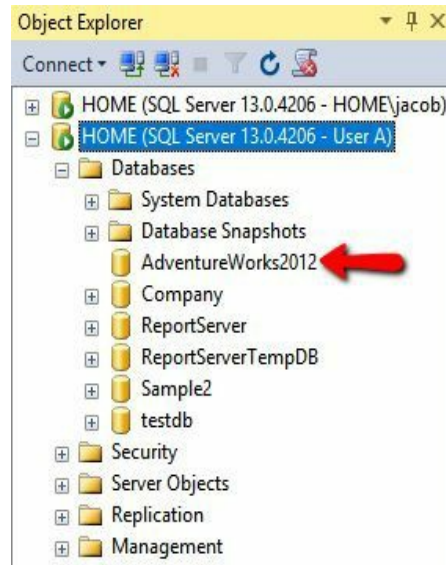


When you see the login window, click on the drop-down menu and select Server Authentication.

Then you can type the username and password to log in.



When you're logged into the user account, you can use the Object Explorer to open a database. Here's how the window looks:



Take note that if you're trying to expand our example database, you'll have just a blank result. Why? Simply because the current user has no access to this particular database. Let's talk more about this in the following section.

## Database Users and Roles

The database user is the user who has access with certain privileges to a specific database or to multiple databases. The database user can also give access to other users in order to use the data inside, and they can also limit that access or even fully restrict it. Take note that in the real world you normally won't give full access to many users. Instead you'd allow them access to a server instance.

Database users also come with a set of database roles that are similar to the server level roles. Here they are:

- **Database owner (db\_owner):** This role provides the user with full control over the database, including the power to delete a database.
- **Database access admin (db\_accessadmin):** This role administers the access levels to a specific database.
- **Database security admin (db\_securityadmin):** This role is in charge of permissions and privileges.

- **Database backup operator (db\_backupoperator):** This role is able to create database backups.
- **Database data writer (db\_datawriter):** This role is assigned to a user who needs access to a certain table where they need to manipulate data.
- **Database data reader:** This role can see the data in the tables.
- **Database DDL admin (db\_ddladmin):** This role can only process **DDL** statements.
- **Db\_denydatawriter** and **db\_denydatareader:** These roles are the opposite of the data writer and data reader roles.

## Assigning Database Roles

The process of assigning database roles and creating users is very similar to assigning server roles and setting up user logins. Again, we can use SQL syntax or the *Management Studio* interface to assign the roles at the database level. Furthermore, you can link a specific user to a login.

Here are some syntax examples which you can use to assign a database user to a specific role and a login.

First, use the database to which you want to add users.

```
USE [AdventureWorks2012]
GO
```

Create a new user.

```
CREATE USER [User A] FOR LOGIN [User A]
GO
```

Specify the database again because we need to assign the role correctly to the preferred user.

```
USE [AdventureWorks2012]
GO
```

You can then change the role of the data writer to add the new user we generated so that they gain the privileges required to manipulate or remove the data in the specified database.

```
ALTER ROLE [db_datawriter] ADD MEMBER [User A]
GO
```

# The LIKE Clause

We use a number of operators together with the **LIKE** clause in order to compare values. Take note that with this clause, the operators also have to include the underscore symbol or the percentage sign. Here are a few syntax examples:

```
SELECT FROM tableName  
WHERE column LIKE 'XXX%'  
SELECT FROM tableName  
WHERE column LIKE '%XXX%'  
SELECT FROM tableName  
WHERE column LIKE 'XXX_'  
SELECT FROM tableName  
WHERE column LIKE '_XXX'  
SELECT FROM tableName  
WHERE column LIKE '_XXX_'
```

When we have more than one condition, they can be merged together with the help of the **AND** and **OR** operators. Now let's go through a series of examples using the **LIKE** clause. Keep in mind that we'll follow the above syntax, but instead of the **XXX**, we're going to have values or strings.

This statement will return a value that starts with 300:

```
WHERE SALARY LIKE '300%'
```

This statement will return a value that has 300 anywhere in it:

```
WHERE SALARY LIKE '%300%'
```

This statement will look for values that have 0 in the second or third position of the value:

```
WHERE SALARY LIKE '_00%'
```

This statement will look for values that are three characters long or more, and that start with a 3.

```
WHERE SALARY LIKE '3_%%'
```

This statement will search for values that end with 3.

```
WHERE SALARY LIKE '%3'
```

This statement will search for values that end with 4 and contain a 3 in the second position of the value.

```
WHERE SALARY LIKE '_3%4'
```

The final statement will look for 5-digit values that start with 3 and end in 4.

```
WHERE SALARY LIKE '3__4'
```

Next, let's see in more detail how this clause is used. We're going to experiment with the help of the **EMPLOYEES** table that contains the following information:

ID	NAME	ADDRESS	AGE	SALARY
2	Mercy	Mercy32	25	3500.00
3	Joel	Joel42	30	4000.00
4	Alice	Alice442	31	2500.00
5	Nicholas	Nich442	45	5000.00
6	Milly	Mil342	32	2000.00
7	Grace	Gra361	35	4000.00

Now, execute the following statement to see all the data entries where the salary value starts with 400:

```
SELECT * FROM EMPLOYEES  
WHERE SALARY LIKE '400%';
```

Here's the result:

ID	NAME	ADDRESS	AGE	SALARY
3	Joel	Joel42	30	4000.00
7	Grace	Gra361	35	4000.00

## The COUNT Function

Let's start with the following example:

```
SELECT COUNT (<expression>)  
FROM table_name;8
```

The expression in our statement can either be about a mathematical operation or the name of a column. Furthermore, you can also calculate the number of

entries found in our table. Here's how you can count the number of rows in the SALES\_REP table:

```
SELECT COUNT(EMP_NAME)
FROM SALES_REP;8
```

This is the result:

COUNT(EMP_NAME)
10

Now, try using (\*) instead of entering the name of the column:

```
SELECT COUNT(*)
FROM SALES_REP;8
```

You'll notice the same result after executing the statement because we don't have any null values in the EMP\_NAME field. However, if this field would have a null value, then we wouldn't add it to the statement that mentions the EMP\_NAME, but instead we would include it in the COUNT() output because the asterisk symbol was used as a parameter.

```
SELECT BRANCH, COUNT(*) FROM SALES_REP
GROUP BY BRANCH;
```

Take note that you can use the COUNT function together with the DISTINCT clause to discover certain entries. For instance, when you need to learn the number of distinct branches that were stored in the table, you can use the following syntax:

```
SELECT COUNT (DISTINCT BRANCH)
FROM SALES_REP;
```

## The AVG Function

Let's start with the following example:

```
SELECT AVG (<expression>)
FROM "table_name";8
```

Remember that the expression can also be a mathematical operation or a column name. Furthermore, a mathematical operation can involve more than one column.



In the following example we can use the **AVG** function in order to determine the value of the average sales from the **Sales\_Rep** table. Here's how it works:

ID	EMP_NAME	SALES	BRANCH
1001	ALAN MARSCH	3000.00	NEW YORK
1098	NEIL BANKS	5400.00	LOS ANGELES
2005	RAIN ALONZO	4000.00	NEW YORK
3008	MARK FIELDING	3555.00	CHICAGO
4356	JENNER BANKS	14600.00	NEW YORK
4810	MAINE ROD	7000.00	NEW YORK
5783	JACK RINGER	6000.00	CHICAGO
6431	MARK TWAIN	10000.00	LOS ANGELES
7543	JACKIE FELTS	3500.00	CHICAGO
	MARK GOTH	5400.00	AUSTIN

```
SELECT AVG(Sales) FROM Sales_Rep;
```

And this is the output:

<b>AVG(SALES)</b>
6245.50

The average value you can see in the output represents the average result from the entire sales information we have in the **Sales\_Rep** table. It is determined by calculating the sum of the sales field and by dividing that value by the number of total sales entries, 10 rows in this case.

Now, let's use the same function in a mathematical operation where we assume a tax of 6.6% of sales and we need to determine the average tax value.

```
SELECT AVG(Sales*.066) FROM Sales_Rep;
```

The mathematical operation is performed first, and only afterward we'll apply the following function:

SELECT Branch, AVG(Sales) FROM Sales_Rep
---

```
GROUP BY Branch;
```

## The ROUND Function

Let's start with the following example:

```
ROUND (expression, [decimal place])
```

Take note that the decimal place refers to the number of points to be returned. So, if we have a -1-value specified, the number will be rounded to the nearest unit number.

Now, let's go through a couple of examples by using the following table as our source of data.

ID	Name	Grade
1	Jack Knight	87.6498
2	Daisy Poult	98.4359
3	James McDuff	97.7853
4	Alicia Stone	89.9753

If you want to approximate to the nearest tenth, the syntax will look like this:

```
SELECT Name, ROUND (Grade, 1) Rounded_Grade FROM Student_Grade;
```

This would be the result:

Name	Rounded_Grade
Jack Knight	87.6
Daisy Poult	98.4
James McDuff	97.8
Alicia Stone	90.0

But what if you want to round down the values without using decimals? To do that, you simply need to add a negative parameter to the function.

```
SELECT Name, ROUND (Grade, -1) Rounded_Grade FROM Student_Grade;
```

This is the output:

Name	Rounded_Grade
Jack Knight	90
Daisy Poult	100
James McDuff	100
Alicia Stone	90

## The SUM Function

The purpose of the **SUM** function is to return the total value. Let's start with the following example:

```
SELECT SUM (<expression>)
FROM "table_name";
```

Again, the expression parameter can be pointed to a mathematical operation or the name of a column. Keep in mind that we can have multiple columns inside the **SELECT** statement as well, without taking into account the column that's mentioned in the function. However, you should use the **GROUP BY** clause as well. Here's an example:

```
SELECT column1, column2, ... columnN, SUM ("column nN + 1")
FROM table_name;
GROUP BY column1, column2, ... column_nameN;8
```

Now we can determine the number of sales with the data from the **Sales\_Rep** table by using this syntax:

```
SELECT SUM(Sales) FROM Sales_Rep;
```

The result will be 62,455.00, which is the sum of the records found in the sales column.

Here's an example where we include a mathematical operation as an argument in the function. We'll assume we have a sales tax of 6.6% that we need to calculate as a total value applied to the sales.

```
SELECT SUM(Sales*.066) FROM Sales_Rep;
```

This is the output:

---

**SUM(sales\*.066)**

---

4122.03

Let's see another example where we use the sum function together with the **GROUP BY** clause to determine the number of sales accomplished by every company branch in our table.

```
SELECT Branch, SUM(Sales) FROM Sales_Rep  
GROUP BY Branch;
```

This is the output:

<b>BRANCH</b>	<b>SUM(Sales)</b>
AUSTIN	5400.00
CHICAGO	13055.00
LOS ANGELES	15400.00
NEW YORK	28600.00

## The MAX() Function

Another useful function is the **MAX** function, which we use to determine the biggest value in a specific expression. Let's take a look at the syntax:

```
SELECT MAX (<expression>)  
FROM table_name;
```

As with the other functions, the expression can be a column or a mathematical operation, which can have several columns. Remember, that the **SELECT** statement can have multiple columns as well, and when that's the case we have to use the **GROUP BY** clause. Here's how it all looks in code:

```
SELECT column1, column2, ... "columnN", MAX (<expression>)  
FROM table_name;  
GROUP BY column1, column2, ... "columnN";
```

To illustrate the use of the syntax, we are going to turn back to our sales table and determine the largest sales amount with the following code:

```
SELECT MAX(Sales) FROM Sales_Rep;
```

This is the output:

<b>MAX(Sales)</b>
-------------------

14600.00

Now let's apply the function to a mathematical operation like we did with the others. We will use the data in our sales table and calculate the highest sales tax value with the following statement:

```
SELECT MAX(Sales*0.066) FROM Sales_Rep;
```

This is the output:

**MAX(sales\*.066)**

963.60

Finally, we can use the function with the **GROUP BY** clause in order to determine the maximum value for each company branch. Here's how:

```
SELECT Branch, MAX(Sales) FROM Sales_Rep GROUP BY Branch;
```

This is the output:

BRANCH	MAX(Sales)
AUSTIN	5400.00
CHICAGO	6000.00
LOS ANGELES	10000.00
NEW YORK	14600.00

## The MIN() Function

This function is essentially the opposite of the **MAX** function, so its application is very similar. With that in mind, let's take a look at an example:

```
SELECT MIN(<expression>)  
FROM table_name;
```

Just like with the **MAX** function, we can perform mathematical operations as well, and the **SELECT** statement can have multiple columns used together with the **GROUP BY** clause. Here's how it looks:

```
SELECT column1, column2, ... "columnN", MIN (<expression>)  
FROM table_name;  
GROUP BY column1, column2, ... "columnN";
```

Here's an example where we use the sales table to determine the lowest value:

```
SELECT MIN(Sales) FROM Sales_Rep;
```

Now, let's determine the lowest sales tax, assuming a tax of 6.6%. As you'll see, it works just like the **MAX** function:

```
SELECT MIN(Sales*0.066) FROM Sales_Rep;
```

Finally, we can use the function together with the **GROUP BY** clause to determine the smallest sales value for each company branch. Here's an example:

```
SELECT Branch, MIN(Sales) FROM Sales_Rep GROUP BY Branch;
```

And here's the output:

<b>BRANCH</b>	<b>MIN(Sales)</b>
AUSTIN	5400.00
CHICAGO	3500.00
LOS ANGELES	5400.00
NEW YORK	3000.00

# CHAPTER 12:

## Dealing with Errors

Creating error-free programs is everyone's unachievable dream. You will eventually encounter various error conditions and if you don't know how to manage them, you're in trouble.

Fortunately, SQL comes with a system that returns information about the errors. This system is `SQLSTATE`, which is a status parameter in essence. Depending on the data within the `SQLSTATE`, you can choose to take various actions to fix the error. For example, you can use the `WHENEVER` clause to take a specific action whenever the set condition is encountered. Furthermore, you can find more information in the diagnostic section.

In this chapter we are going to focus on how to figure out the source of an error and how to handle it appropriately.

### **SQLSTATE**

`SQLSTATE` is a string of five characters in length, containing only the uppercase letters from A to Z and numbers from 0 to 9. Any other character is invalid. This string is used to determine a number of anomalous conditions and it can represent two groups of codes. One is a class code that contains the status at the end of a statement's execution, with the code being only two characters in length. The other is a subclass code with three characters in length and it offers us more information about the statement execution. In other words, the status can either tell us that the statement ran successfully or that an error was found, and the subclass code gives us more details.

The class codes follow a standard and those that start with any letter from A to H, or numbers from 0 to 4, will represent the same error no matter which SQL implementation you're using. All the other codes can be customized by the database management system developers because they're needed to determine specific conditions that only appear in certain implementations. However, you shouldn't worry too much about the custom codes, even though you should find them in your database management system's documentation, because developers try their best to stick to the standard ones with minor exceptions.

Here are some of the SQLSTATE class codes you'll encounter the most:

1. Code 00: This means the statement was executed successfully.
2. Code 01: The execution was a success, but there's one warning.
3. Code 02: This code brings up a "no data condition" warning.
4. From 03 and beyond: Any class code other than the first three means that the statement wasn't successfully executed and you need to look into the details to fix the error.

The class codes you get will tell you what the problem is and what options you might have for solving it.

## **The WHENEVER Clause**

Knowing that something went wrong isn't enough. The problem is that when an error is encountered, the application might continue running other statements as if it's pretending that everything is fine. We need to stop it from running, determine the error, and fix it. If we can't figure out a solution, then the program should at least tell the user that an error was encountered and that the program is going to be terminated. This is where execution exceptions come in with the help of the WHENEVER clause. Here's how the syntax looks:

WHENEVER condition action;

A condition can be something like "SQLERROR" and the action can be an instruction such as "CONTINUE", or "GOTO" and specify an address. The SQLERROR will be encountered if the class code of the SQLSTATE is something other than the first three we discussed above. If we choose the



CONTINUE action, then the execution will go on like nothing happened. However, by using the GOTO action together with an address, we can instruct the program to move the execution to a specific location. This location, or address, can contain a conditional statement that will verify the SQLSTATE so that we can run a specific action depending on the encountered state. Here are a couple of examples:

```
WHenever SQLERROR GOTO annoying_errors;
```

```
WHenever NOT FOUND CONTINUE;
```

And here is a final example where we use the GOTO option with a conditional:

```
IF SQLSTATE <> '00000'  
  AND SQLSTATE <> '00001'  
  AND SQLSTATE <> '00002'
```

```
THEN GOTO annoying_errors;
```

## Diagnostics

Even though you can use SQLSTATE to get most of the clues you need about a failure, the information is still not always as detailed as we'd like. Fortunately, SQL has a number of details that you can access in the diagnostics area. You can find the most recent error at the top of the list, accompanied by all the status data including multiple warnings that are associated.

The diagnostics area is built from two elements. First you have the header that holds the general information about the execution of a statement, and then you have the detail area that contains data about every bit of generated code.

Here are some of the objects found in the header section and what they mean:

1. **NUMBER:** This field tells us how many detail areas contain information about an error.
2. **ROW\_COUNT:** Here you'll see the total number of rows that were affected.
3. **COMMAND\_FUNCTION:** In this field you can find data about

the SQL statement the program executed.

4. `COMMAND_FUNCTION_CODE`: Each statement receives a code that's associated with it.
5. `DYNAMIC_FUNCTION`: This is where you'll find the dynamic statement.
6. `DYNAMIC_FUNCTION_CODE`: The dynamic statement also received an associated numeric code.
7. `MORE`: In this field you'll see a "y" or "n" as a result. The first one tells us whether there are more records than the detail section can contain. If you encounter the second option, then all records have been displayed.

Next, we have the detail section where we find the precious information on each error and warning:

1. `CONDITION_NUMBER`: This is where you'll find the sequence number. In other words, if a statement results in five different items that are stored in five separate detail sections, the condition number will be equal to 5. This is important because this number is used to extract a certain detail section so that you can perform an analysis by using the `GET DIAGNOSTICS` statement together with the condition number.
2. `RETURNED_SQLSTATE`: In this section you'll find the returned `SQLSTATE` value that was the reason why the detail section was filled in the first place. Based on this value we can look over the `CLASS_ORIGIN` to find its source.
3. `CLASS_ORIGIN`: As mentioned, this is what provides us with the source of the class code value. If the value is defined by the SQL standard, the origin should be ISO 9075. However, the database management system can also define the value and in that case the origin will contain a string that identifies the source of the DBMS instead.
4. `SUBCLASS_ORIGIN`: As the name suggests, this is where you'll learn the source of the subclass code's value that's obtained by the `SQLSTATE`.

Let's take a look at an example. Assume that the `SQLSTATE` returns 22012

and the values show that we're within a range of typical SQLSTATEs. This means that no matter what SQL implementation we're dealing with, the code means the same thing. But, if we have a 22500 code, only the '22' characters are found within the standard and they point us towards an exception. On the other hand, the last three characters depend on the definition of the implementation. So, how do you figure out what caused the error and what the characters in the code mean? You'll have to check the documentation of your SQL implementation. Once you do that, you can learn more about the error condition by analyzing the class origin and the subclass origin.

## Exceptions

There are three main methods of handling an exception. But first, remember that you'll encounter an exception condition when the SQLSTATE contains a value that isn't equal to 00000, 00001, and 00002.

One option is to simply give back control to the main procedure that executed the subprocess that led to the exception.

We can also use the WHENEVER clause instead by declaring another action or by including a specific exception handling process that branches out to certain actions based on the events.

Finally, we can also use a compound SQL statement to deal with the exception, namely a statement that contains multiple statements that are declared in between BEGIN and END. Let's look at a common example:

```
BEGIN
DECLARE ValueOutOfRange EXCEPTION FOR SQLSTATE'73003' ;
INSERT INTO EMPLOYEES
(Accountants)
VALUES
(:acc) ;
SIGNAL ValueOutOfRange ;
MESSAGE 'Executing a new value.'
EXCEPTION
WHEN ValueOutOfRange THEN
MESSAGE 'Handling the accountant range error' ;
WHEN OTHERS THEN
RESIGNAL ;
END
```

Take note that when using a DECLARE statement we can name the SQLSTATE values. Furthermore, chances are that the INSERT statement will be the source of the exception.

In our example, if we have the “acc” value higher than the max value of the data object, we’re going to have an SQLSTATE of 73003. Then we use a SIGNAL statement to clear the diagnostics section and signal the exception. This is where the returned SQLSTATE from the diagnostics area will be set to that of the exception. However, if nothing happens, the “Executing a new value” statement will be executed. When the exception is raised, that statement is ignored, and the code continues to the EXEPTION section.

In this scenario we have a ‘value out of range’ type of exception (really common) and therefore we execute the range handling statement. Finally, we have a RESIGNAL statement just in case we aren’t dealing with a value out of range exception. This is used to transfer the execution commands back to the parent procedure in case there’s more code fitted for other types of exceptions.

# CONCLUSION

As the world continues to exponentially generate immense amounts of data, more and more database managers and analysts are needed to handle it all. We are heading towards a world where Big Data needs to be processed and secured, while every small business wants to learn from the numbers they generate. In either case, SQL is right at the center, allowing us to manage data efficiently and securely.

You've gone through a great deal of information, and at times it may have felt hectic, but that's how the learning process feels no matter what you study. So even when some SQL concepts feel a little dry, you need to push through. With the data in this book and with plenty of practice, you should be able to perform most of the tasks that involve SQL and databases. Make sure to go back and check out the areas you haven't fully understood. If necessary, you can do some research on your own as this is a popular field with rich online communities.

## The Next Steps

Before you start exploring more advanced SQL topics, you should make sure you master the fundamentals. The best way to achieve this is through practical work on real databases. You can easily go online and find sample databases to play with and try out all the operations and procedures we discussed on your own. For instance, you can look at the *Chinook* or *Northwind* databases and get started with them.

The Chinook sample database is actually based on the Northwind database, and it represents a digital media store with data regarding artists, media albums, customers, invoices, and other tables. All you need to do is download the database by clicking on the “clone or download” arrow and selecting “download ZIP.” Once you’ve finished downloading the file, you can extract everything from the zip archive and drop the SQL files into your database management tool. The Northwind sample database represents the kind of data a small business would store.

Start working with database samples that mimic real world data and don’t just practice SQL queries because you also need to focus on understanding the actual data. Working with SQL isn’t all about management and writing SQL code. You must understand what’s in the database as well.

Furthermore, you should work on database security because keeping the data safe is one of your top priorities. Master SQL’s security scheme and learn how to manage the actions that a specific user can see or perform and work with authorization identifiers.

# REFERENCES

- 1keydata.com. (2019). *SQL - RENAME COLUMN* | 1Keydata. [online] Available at: <https://www.1keydata.com/sql/alter-table-rename-column.html> [Accessed 3 Feb. 2019].
- 1keydata.com. (2019). *SQL - CREATE VIEW Statement* | 1Keydata. [online] Available at: <https://www.1keydata.com/sql/sql-create-view.html> [Accessed 3 Feb. 2019].
- Chartio. (2019). *How to Alter a Column from Null to Not Null in SQL Server*. [online] Available at: <https://chartio.com/resources/tutorials/how-to-alter-a-column-from-null-to-not-null-in-sql-server/> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Primary and Foreign Key Constraints - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Create Check Constraints - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-check-constraints?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Create Primary Keys - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-primary-keys?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Delete Tables (Database Engine) - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/delete-tables-database-engine?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Modify Columns (Database Engine) - SQL*

- Server. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/modify-columns-database-engine?view=sql-server-2017> [Accessed 3 Feb. 2019].
- query?, H., M, D., K., R., Singraul, D., Singh, A. and kor, p. (2019). *How to change a table name using an SQL query?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/886786/how-to-change-a-table-name-using-an-sql-query> [Accessed 3 Feb. 2019].
- Sites.google.com. (2019). *DDL Commands - Create - Drop - Alter - Rename - Truncate - Programming Languages*. [online] Available at: <https://sites.google.com/site/prgimr/sql/ddl-commands---create---drop--alter> [Accessed 3 Feb. 2019].
- SQL Joins Explained. (2019). *SQL Join Types — SQL Joins Explained*. [online] Available at: <http://www.sql-join.com/sql-join-types/> [Accessed 3 Feb. 2019].
- Stanek, W. (2010). *Microsoft SQL Server 2008 Administrator's Pocket Consultant*. Microsoft Press.
- Techonthenet.com. (2019). *SQL: ALTER TABLE Statement*. [online] Available at: [https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php) [Accessed 3 Feb. 2019].
- Techonthenet.com. (2019). *SQL: GROUP BY Clause*. [online] Available at: [https://www.techonthenet.com/sql/group\\_by.php](https://www.techonthenet.com/sql/group_by.php) [Accessed 3 Feb. 2019].
- Techonthenet.com. (2019). *SQL: UNION ALL Operator*. [online] Available at: [https://www.techonthenet.com/sql/union\\_all.php](https://www.techonthenet.com/sql/union_all.php) [Accessed 3 Feb. 2019].
- www.tutorialspoint.com. (2019). *SQL DEFAULT Constraint*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-default.htm> [Accessed 3 Feb. 2019].
- www.tutorialspoint.com. (2019). *SQL INDEX Constraint*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-index.htm> [Accessed 3 Feb. 2019].
- www.tutorialspoint.com. (2019). *SQL INNER JOINS*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-inner-joins.htm> [Accessed 3 Feb. 2019].



www.tutorialspoint.com. (2019). *SQL LIKE Clause*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-like-clause.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL NOT NULL Constraint*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-not-null.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL TOP, LIMIT or ROWNUM Clause*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-top-clause.htm> [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL INNER JOIN Keyword*. [online] Available at: [https://www.w3schools.com/sql/sql\\_join\\_inner.asp](https://www.w3schools.com/sql/sql_join_inner.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL LEFT JOIN Keyword*. [online] Available at: [https://www.w3schools.com/sql/sql\\_join\\_left.asp](https://www.w3schools.com/sql/sql_join_left.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL NOT NULL Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_notnull.asp](https://www.w3schools.com/sql/sql_notnull.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL NOT NULL Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_notnull.asp](https://www.w3schools.com/sql/sql_notnull.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL PRIMARY KEY Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_primarykey.asp](https://www.w3schools.com/sql/sql_primarykey.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL RIGHT JOIN Keyword*. [online] Available at: [https://www.w3schools.com/sql/sql\\_join\\_right.asp](https://www.w3schools.com/sql/sql_join_right.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL UNION Operator*. [online] Available at: [https://www.w3schools.com/sql/sql\\_union.asp](https://www.w3schools.com/sql/sql_union.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL UNION Operator*. [online] Available at: [https://www.w3schools.com/sql/sql\\_union.asp](https://www.w3schools.com/sql/sql_union.asp) [Accessed 3 Feb. 2019].

W3schools.com. (2019). *SQL UNIQUE Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_unique.asp](https://www.w3schools.com/sql/sql_unique.asp) [Accessed 3 Feb. 2019].

2019].

www.tutorialspoint.com. (2019). *SQL Primary Key*. [online] Available at:  
<https://www.tutorialspoint.com/sql/sql-primary-key.htm> [Accessed 3 Feb. 2019].

# SQL

**The Ultimate Intermediate Guide to Learning SQL  
Programming Step by Step**

**© Copyright 2022 - All rights reserved.**

The content contained within this book may not be reproduced, duplicated, or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

**Legal Notice:**

This book is copyright protected. It is only for personal use. You cannot amend, distribute, sell, use, quote, or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaged in the rendering of legal, financial, medical, or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of the information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

# Table of Contents

## [Chapter 1:](#)

[Basics of SQL](#)

[Data and Databases](#)

[Data](#)

[Databases](#)

[Relational Database](#)

[Web/Cloud-Based Database Systems](#)

[Client/Server Technology](#)

[The Elements of an SQL Database](#)

[Tables](#)

[Fields](#)

[Records](#)

[Database Schemas](#)

[SQL Server](#)

## [Chapter 2:](#)

[Installing and Configuring MySQL](#)

[Downloading, Installing, and the Initial Setup of MySQL on a Microsoft Windows Computer System](#)

[NOTE:](#)

[Downloading, Installing, and the Initial Setup of MySQL on a Mac Computer System](#)

## [Chapter 3:](#)

[Getting Started With SQL](#)

[MySQL Screen](#)

[Working With MySQL Databases](#)

[Creating a Database with MySQL Workbench](#)

[Deleting a Database with MySQL Workbench](#)

[Working With Tables](#)

[Types of Storage Engines Supported by SQL](#)

[Archive](#)

[Blackhole](#)

[CSV](#)

[InnoDB](#)

[MyISAM](#)

[NDB](#)

[Creating a Table](#)

[Adding Data to a Table](#)

[Exercise 1](#)

[Viewing Data in a Table](#)

[Chapter 4:](#)

[Data Types](#)

[SQL Data Type Categories](#)

[Approximate Numeric Data Types](#)

[Default Data Type Values](#)

[Chapter 5:](#)

[SQL Statements and Clauses](#)

[SQL Statements](#)

[SQL Clauses](#)

[Chapter 6:](#)

[SQL Expressions, Functions, and Operators](#)

[Expressions](#)

[Conditional Value Expressions](#)

[Datetime Value Expressions](#)

[Interval Value Expressions](#)

[Numeric Value Expressions](#)

[String Value Expressions](#)

[Operators](#)

[Arithmetic](#)

[Comparison](#)

[Logical](#)

[Chapter 7:](#)

[Working With Constraints](#)

[Commonly Used SQL Constraints](#)

[Adding Constraints](#)

[Altering Constraints](#)

[Exercise 2](#)

[The HR Database](#)

[Chapter 8:](#)

[Joins, Unions, Ordering, Grouping, and Alias](#)

[Types of Joins](#)

[Joining More Than Two Tables](#)

[UNION](#)

[Order](#)

[Group By](#)

[Alias](#)

[Chapter 9:](#)

[Stored Procedures](#)

[The Advantages of Using Stored Procedures](#)

[Creating Stored Procedures](#)

[Executing Stored Procedures](#)

[Chapter 10:](#)

[Views, Index, Truncate, Top, Wildcards, and Triggers](#)

[Encrypting a View](#)

[Creating a View](#)

[Index](#)

[Table Indexes](#)

[Single-Column Indexes](#)

[Composite Index](#)

[Implicit Index](#)

[Unique Index](#)

[Deleting an Index](#)

[Indexing Tips](#)

[Truncate](#)

[Top](#)

[Wildcards](#)

[Triggers](#)

[Trigger Syntax](#)

[Chapter 11:](#)

[Pivoting Tables in MySQL](#)

[Create a Products Sales Database](#)

[Chapter 12:](#)

[Clone Tables](#)

[Why Use a Clone Table?](#)

[Chapter 13:](#)

[Security](#)

[Components of Database Security](#)

[Three-Class Security Model](#)

[Schemas](#)

[Server Roles](#)

[Logins](#)

[Mixed Mode Authentication](#)

[Database Roles](#)

[Encryption](#)

[Master Keys](#)

[Transparent Data Encryption \(TDE\)](#)

[Chapter 14:](#)

[SQL Injections](#)

[Hacking Scenario](#)

[Chapter 15:](#)

[Fine-Tuning](#)

[SQL Tuning Tools](#)

[Chapter 16:](#)

[Working With SSMS](#)

[Downloading SQL Server Management Studio \(SSMS\)](#)

[Connect to SQL Server with SSMS](#)

[The Basics and Features of SSMS](#)

[Managing Connections](#)

[Choosing Your Database](#)

[Chapter 17:](#)

[Database Administration](#)

[Maintenance Plan](#)

[Defining Tasks for the Maintenance Plan](#)

[Running the Maintenance Plan](#)

[Backup and Recovery](#)

[Database Backup](#)

[Performing a Backup](#)

[Attaching and Detaching Databases](#)

[Chapter 18:](#)

[Deadlocks](#)

[Chapter 19:](#)

[Normalization of Your Data](#)

[How to Normalize the Database](#)

[Raw Databases](#)

[Logical Design](#)



[The Needs of the End-User](#)

[Data Repetition](#)

[Normal Forms](#)

[Naming Conventions](#)

[Benefits of Normalizing Your Database](#)

[Denormalization](#)

[Database Normal Forms](#)

[First Normal Form \(1NF\)](#)

[Second Normal Form \(2NF\)](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Fourth Normal Form \(4NF\)](#)

[Fifth Normal Form \(5NF\)](#)

[Chapter 19:](#)

[Real-World Uses](#)

[SQL in an Application](#)

[Database Locks](#)

[Conclusion](#)

[References](#)

# CHAPTER 1:

## Basics of SQL

In today's environment, there are vast quantities of data that are collected nearly every second of the day. To be able to make this data useful it needs to be organized, edited, and structured in certain ways depending on the needs of the organization using it.

To understand how data can be manipulated using SQL, you first need to understand how the process all fits together. This chapter briefly outlines all the parts that fit together to create a workable database system.

## Data and Databases

### Data

Data is a collection of values, facts, and figures that has been gathered for some purpose. For instance, every ten years or so local or national governments collect data on residents within their cities or burroughs—census data. This data is gathered to help with various demographic conditions, city planners, and to aid government departments in understand various economic or social structures.

Data is only considered viable if it is meaningful. Meaningful data can come in various forms and come from different sources such as:

- Agricultural data
- Blogs
- Cultural data
- Educational data

- Environmental data
- Financial data
- Geographical data
- Meteorological data
- Scientific data
- Social media
- Statistical data
- Transport data

## **Databases**

A database is a library, or storage area, where the collected data is stored. A database is bidirectional in that it can receive input and return the requested information. When data is imputed from a user, it is processed, converted, and then used as raw data/schema.

People use databases nearly every day without even realizing they're using them. For example, when you store a new contact on your phone, you are using a database. Here you would store information such as the person's name, surname, work, phone number(s), maybe even their physical addresses. When you look up that person's number, you are scrolling through a phonebook database to find and retrieve a phone number.

There are different types of databases:

- Cloud database
- Centralized database
- Distributed database
- Graph database
- NoSQL database
- Object-oriented database
- Operational database
- Relational database

## **Relational Database**

Simply put, a relational database is a database that stores information and organizes it as a collection of tables with related data points. These related data points are connected by a unique ID called the Primary Key or Key.

Relational databases store information in tabular form which makes data more flexible to store, access, and manipulate. To better understand this, take a look at the tables below for a simple name, phone number, and address book:

#### Simple Name, Address, & Phone Number Database

**Database Tables**

PB Name			
ID	First Name	Last Name	Click to Add
3	Peter	Grimm	
4	Stacy	Apple	
(New)			

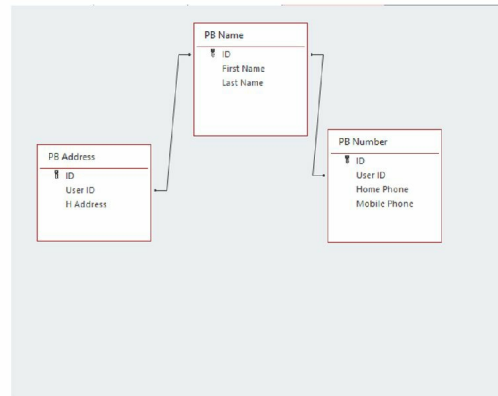
  

PB Number				
ID	User ID	Home Phon	Mobile Phon	Click to Add
2	3	1555555555	1777777770	
4	4	707070700	65555550	
0	0	0	0	

PB Address		
ID	User ID	H Address
2	3	122 This Way Dr
3	4	444 That Way Dr
(New)	0	

**Database Relationships**



The phone numbers and address lists are joined to the names table by the User ID key. The above is a very basic example. Another example is that of a customer list and ordering system. Each customer will get a unique ID Key. This is the key that will be used to identify which customer is placing the order. The ID key is what forms the relationship bond between the two tables. There are many relational database management programs on the market today. Some of the more popular relational databases on the market are:

- Azure SQL
- IMB DB2
- Informix
- MariaDB
- Microsoft Access
- MongoDB
- MySQL
- Oracle
- PostgreSQL
- SQLite

- SQL Server

## **Web/Cloud-Based Database Systems**

A web or cloud database is a database that is accessed through the Internet. More and more companies as well as private individuals are moving towards web-based systems and applications.

Some of the top cloud-based relational databases include:

- Amazon Relational Database
- Couchbase
- DataStax
- Google Cloud SQL
- IBM Db2 on Cloud
- Microsoft Azure SQL Database
- MongoDB Atlas
- Oracle Database Cloud

## **Client/Server Technology**

Historically, data was stored on large systems known as mainframe computers. To access the information on these systems the user would use a dumb terminal. Most of these terminals had a small screen and keyboard. They were heavy and the screen was black with either green or blueprint. They relied on the mainframe being accessible and drew all the information from it. If the mainframe was down, the terminals would not work.

Although there are still mainframes around, file servers have become the normal system storage solutions. A file server is either one powerful computer or a data center of powerful computers. These are the systems that not only store files, they also run various programs and allow for shared resources.

The end-user that needs to access the network does so through a network client. This is a desktop PC, a laptop, notebook, tablet, etc. For each program/database there will be the backend, which runs on the server, and the front-end, which runs on the client. The client accesses the data on the server by making a service request to the server. The server will process the request and respond. This is a client-server relationship.

The client is reliant on the server to manage, process, and provide information. However, most programs these days are resilient and allow the client to run independently from the server. If the server is down for any reason, depending on the system, the client may still be able to work. When the server is restored the information will update from the client to the server.

There are some applications, like web browsers, or web/cloud-based databases that are reliant on the back-end servers being up.

## The Elements of an SQL Database

An SQL database consists of the following elements:

### Tables

Tables or datasheets are much like a spreadsheet that holds specific related information. A database is made up of many different, but related tables. For example, in a simple ordering system you could have the following tables:

- Customer table — This table would hold all the customer's information.
- Stock Items table — This table would hold all the items held in stock.
- Orders table — This would hold all the customer's orders of the stock being sold.
- Invoices table — This would hold the invoices that were generated for the customers.
- Payments table — This would hold the payments that were made by the customers.

A table has to have a unique name within a database.

Each table will have a specific or set number of columns. Each column will have a unique identifying name that relates to the data and data type being stored in it.

Cells are an **attribute**. Each attribute will be defined by the name of the column and the data type to be stored in the column.

Each row will have a unique row ID or key that is used to identify the record or data in that row.

A row in a database is called a **tuple**. Each table will have an undefined number of rows or records that can and will grow as more records are added to the database.

Where the column and row meet is called a cell.

## Fields

Each table will have several columns depending on the type of data the table was being used to collect. For instance, in the simple ordering system:

- Customer table you may have columns/fields for:
  - First Name
  - Last Name
  - Telephone
  - Address
- Stock Items you have columns/fields for:
  - Stock code
  - Item
  - Description
  - Price

Fields, also known as attributes, get set up to accommodate the type of data that is to be stored in the field. For instance:

- Stock codes would be an alphanumeric field
- Names would be a text field
- Quantity would be a numeric field
- Price would be a currency

There are many different types of fields, and each would be set up to not only hold but manage and manipulate the data. Names would need to be set as text to be able to effectively sort through them. Numbers have a few purposes such as codes, quantities, percentages, and currency. Each of these is used and calculated in a certain way and needs to be established when the database is set up.

## Records

A table or datasheet also has a set of rows. These rows are the records or tuples and are what group the fields together. Each row will make up a

collection of related information about the row's unique ID or key. For example, in a customer table, a single row would contain the customer's customer code, first name, last name, address, contact number, and so on.

## **Database Schemas**

Every database contains what is called an object ownership group. These groups are known as a schema. This is the blueprint of the logical view as to how the database is structured and the relationships between the data.

Schemas are also used to form data constraints which help to authorize or prevent access for certain users or groups to various database tables, fields, and records. It is like a file that contains pertinent information with regard to certain parts of a database.

## **SQL Server**

The actual SQL Server is a database engine that processes transactions such as creating databases, creating and manipulating tables, and handling the data contained in them.

It is also a Relationship Database Management System that manages the databases that are housed within its system.

SQL Server not only manages the data on the system, it also ensure that the database is:

- Secure
- Performs well
- Ensures data integrity
- Ensures data reliability
- Allows multiple concurrent access to the database

To use an SQL database, you need to have an instance of the SQL server running on the system that houses the SQL database. There can be more than one instance of an SQL server running on a system at one time.

Each SQL Server instance can contain one or more databases. Each database running on an SQL server will have a set of stored procedures, views, tables, and records. To access the SQL database, a user would need to log in with a user ID.



Each user ID will be granted access according to the specific task they need to perform within the database. Some databases do not require user logins. There are many programs or management tools that can be used to access SQL databases. These tools include:

- Azure Data Studio
- DataGrip
- DCIM
- RazorSQL
- SQL Server Management Studio (SSMS).
- Squirrel SQL

You can manage the SQL Server engine using management tools such as:

- SQL Server Management Studio (SSMS)
- Business Intelligence Developer
- Configuration Management
- SQL Profiler

# CHAPTER 2:

## **Installing and Configuring MySQL**

For this book, you will be using MySQL. MySQL is a relational database system that uses SQL syntax to facilitate the manipulation and management of SQL databases. It is one of the most popular RDBMS systems because it is free to download, easy to configure, and easy to use.

### **Downloading, Installing, and the Initial Setup of MySQL on a Microsoft Windows Computer System**

From the MySQL website [www.mysql.com](http://www.mysql.com) select the “DOWNLOADS” menu option.

From the “DOWNLOADS” page select MySQL Community (GPL) Downloads, which you will find right at the bottom of the page.

On the next page select the “MySQL Community Server” option.

From the “Select Operating Systems” option, select the “Microsoft Windows” operating system from the drop-down selection box.

Click on the “MySQL Installer for Windows.”

On the next page choose and “Download” the first installer on the page “Windows (x86, 32-bit), MSI Installer”.

On the next page, there will be a few extra options to either “Login” or “Sign Up.” Click on “No thanks, just start my download” near the bottom of the page.

The installer will download it.

Once it has downloaded, click on the “mysql-installer-web-community-8.0.21.0.msi” file in the “Downloads” folder.

If a pop-up box appears that asks “Do you want to allow this app to make changes,” click on “Allow.”

You may also be prompted to run an upgrade. Allow the system to run an upgrade.

The next screen is “Choosing a Setup Type.”

It defaults to “Developer Default.” Leave it at this option and click the “Next” button.

## **NOTE:**

- If you get a screen that lists some MySQL install instances with an error about “Microsoft Visual C++ Redistributable 2019 not installed,” exit the installation.
- Go to the following website: <https://www.itechtics.com/microsoft-visual-c-redistributable-versions-direct-download-links/#6-microsoft-visual-c-redistributable-2019>.
- Download the version the machine is asking for and restart the MySQL Community installation.

The “Check Requirements” screen has an option to connect to Python. Leave this option unchecked and click on the “Next” button. It is good to note that at this screen it may list some other packages such as “MySQL For Excel” or “MySQL for Visual Studio.” Do not install any of these packages.

You will get a pop-up box that says, “One or more product requirements have not been satisfied.” Click on “Yes.”

The “Installation” screen will have a whole list of products that need to be installed on your system.

Click “Execute.”

This will take a few minutes to install, and each application that is being installed will have a progress count next to it. Let the installation finish.

The installation is complete when all the products to be installed have a green checkmark next to them.

Click on “Next.”

This takes you to the “Product Configuration” screen.

This screen lists products that have been installed that are now ready to be configured.

Click on “Next.”

The next screen is called either “Group Replication” or “High Availability.”

It will have the option to choose a “Standalone MySQL Server / Classic MySQL Replication” or “InnoDB Cluster.”

Leave it on the default option which is the “Standalone MySQL Server / Classic MySQL Replication” option and click “Next.”

At the “Type of Networking” options screen leave all the settings as they are (on the default settings) and click “Next.”

At the “Authentication Method” screen leave the choice as the default method for authentication, “Use Strong Password Encryption for Authentication (RECOMMENDED)” and click on “Next.”

At the “Accounts and Roles” screen you will need to set a password for the “root” user. Choose a strong password that you will remember. The “root” user is the admin user for MySQL. When you have created a password, leave the rest of the options as default and click “Next.”

At the “Windows Service” screen you will leave all the default options and click on “Next.”

At the “Apply Configuration” screen click on the “Execute” button and the system will apply all the settings. This will take a few minutes.

When the configuration is done there will be green check marks next to all the configuration steps. Click on “Finish.”

At the “Product Configuration” screen click “Next.”

At the “MySQL Router Configuration” screen, leave all the default settings

and click “Finish.”

At the “Connect To Server” screen, leave all the settings as the default settings. You do need to type in the root password you have just created in the “Password” box. Click on the “Check” box to see if it is correct. If you get a green check next to the “Check” button, click “Next.”

At the “Apply Configuration” screen, click on “Execute.”

When the configuration has successfully finished, click on “Finish.”

At the next “Product Configuration” screen click on “Next.”

The “Installation Complete” screen will appear and there should be two checked options on the screen which are “Start MySQL Workbench after Setup” and “Start MySQL Shell after Setup.” Leave both of these checked and click on “Finish.”

Two screens will open:

- A command prompt — In this tutorial, you will not be working with the command prompt to execute instructions so you can close this screen.
- “Welcome to MySQL Workbench” — You will be using MySQL Workbench, so you will need to leave this screen open for now.

On the “Welcome to MySQL Workbench” screen you should see MySQL Connections near the bottom of the page.

There should be a default connection:

- Local instance MySQL80
- root
- localhost: 3306

If you do not have the above or similar you can create it by:

Next to “MySQL Connections” click on the + icon.

You will be taken to the “Setup New Connection” screen.

- Connection Name: Local Connection (this is a name you can set up)
- Connection Method: Standard (TCP/IP) (this is the default)

connection type)

- Hostname: 127.0.01 (this is the localhost address)
- Port: 3306 (this is the default SQL port)
- Username: root
- Password: you will need to click on the “Store in Keychain” and type in the password you set up during the installation.
- Default Schema: Leave this blank

Test the connection by clicking on the “Test Connection” button at the bottom right-hand side of the screen.

You should see the “Successfully made the MySQL connection” screen.

Click on “OK” on the pop-up screen and again on the bottom right-hand side of the screen.

The next screen will be the “Welcome to MySQL Workbench” screen where you should now see the connection you set up under “MySQL Connections.”

Whenever you open MySQL Workbench you will see this connection. To access the local MySQL server you will need to click on this connection.

The next screen will be the MySQL Workbench, and you will have completed setting up MySQL.

## **Downloading, Installing, and the Initial Setup of MySQL on a Mac Computer System**

From the MySQL website [www.mysql.com](http://www.mysql.com) and select the “DOWNLOADS” menu option.

From the “DOWNLOADS” page select MySQL Community (GPL) Downloads, which you will find right at the bottom of the page.

On the next page click on “MySQL Community Server” option.

From the “Select Operating Systems” option, select the “macOS” operating system from the drop-down selection box.

Download the first option in the list “macOS 10.15 (x86, 64-bit), DMG Archive”.

On the next page, there will be a few additional extra options to either

“Login” or “Sign Up”. Click on “No thanks, just start my download” near the bottom of the page.

The .dmg file will download.

Once the .dmg file has downloaded, open the directory where the file has been downloaded to.

Double-click on the “mysql-8.0.15-macos-x86\_64.pkg” file which will launch the installation.

Click on “Continue” when the warning/installation box appears on the screen.

Click on “Continue” over the next few screens until you get to the Licence agreement.

Click “Agree” to the license.

Click “Continue” once again on the Installation type screen.

This will begin the installation of MySQL.

If you get a message “Installer is trying to install new software” and asks for a password, you must enter the password you use to access your computer.

The software will install.

You will get to the “Configure MySQL Server” screen. Here you will need to choose the first option “Use Strong Password Encryption”. Click “Next”.

In the “Please enter a password for the “root” user” you will need to enter a memorable password. This is the root user password, so make sure you can remember it. When you have entered your password click “Continue”.

MySQL will continue to install and start the initial database setup.

You may need to enter your macOS password one more time.

When the “The installation was completed successfully” screen appears, click on the “Close” button.

Once the MySQL Community Server has downloaded you will need to download MySQL Workbench. This is the API interface software that allows you to communicate with the MySQL Community Server.

Go back to the “DOWNLOADS” page and select MySQL Community (GPL)

Downloads once again.

On the next page select “MySQL Workbench”.

From the “Select Operating Systems” option, select the “macOS” operating system from the drop-down selection box.

Download the first option which is “macOS (x86, 64-bit), DMG Archive”.

On the next page, there will be a few extra options to either “Login” or “Sign Up”. Click on “No thanks, just start my download” near the bottom of the page.

The .dmg file will download.

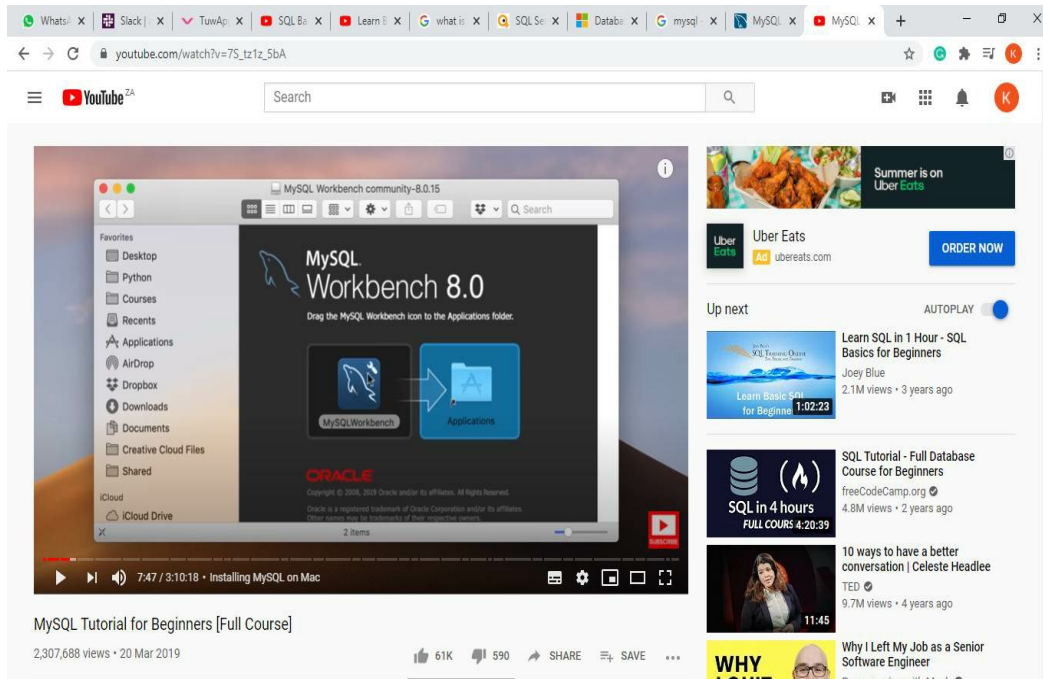
Once the .dmg file has downloaded, open the directory where the file has been downloaded to.

Double-click on the “mysql-workbench-community-8.0.15-macos-x86\_64.dmg” file which will launch the installation.

This screen will be a bit different. You get to the “MySQL Workbench 8.0” screen. Beneath the title, there will be instructions “Drag the MySQL Workbench icon to the Applications folder”.

Drag the MySQL Workbench Icon from the left side (in the small black box) to the blue box with the Application folder in it.





This will copy the file into the Applications folder.

Once the system has finished copying over the application, press the command spacebar to search for “MySQL Workbench”.

Open the application.

You may get a warning box “MySQLWorkbench is an app downloaded from the Internet. Are you sure you want to open it?”

Click “Open”.

On the next screen, you should see MySQL Connections near the bottom of the screen. There should be a default connection.

- Local instance 3306
- root
- localhost: 3306

If you do not have the above or similar you can create it by:

Next to “MySQL Connections” click on the + icon

You will be taken to the “Setup New Connection” screen

- Connection Name: Local Connection (this is a name you can set up)

- Connection Method: Standard (TCP/IP) (this is the default connection type)
- Hostname: 127.0.0.1 (this is the localhost address)
- Port: 3306 (this is the default SQL port)
- Username: root
- Password: you will need to click on the “Store in Keychain” and type in the password you set up during the installation.
- Default Schema: Leave this blank

Test the connection by clicking on the “Test Connection” button at the bottom right-hand side of the screen.

You should see the “Successfully made the MySQL connection” screen.

Click on “OK” on the pop-up screen and again on the bottom right-hand side of the screen.

The next screen will be the “Welcome to MySQL Workbench” screen where you should now see the connection you set up under “MySQL Connections”.

Whenever you open MySQL Workbench you will see this connection. To access the local MySQL server you will need to click on this connection.

The next screen will be the MySQL Workbench and you will have completed setting up MySQL.

# CHAPTER 3:

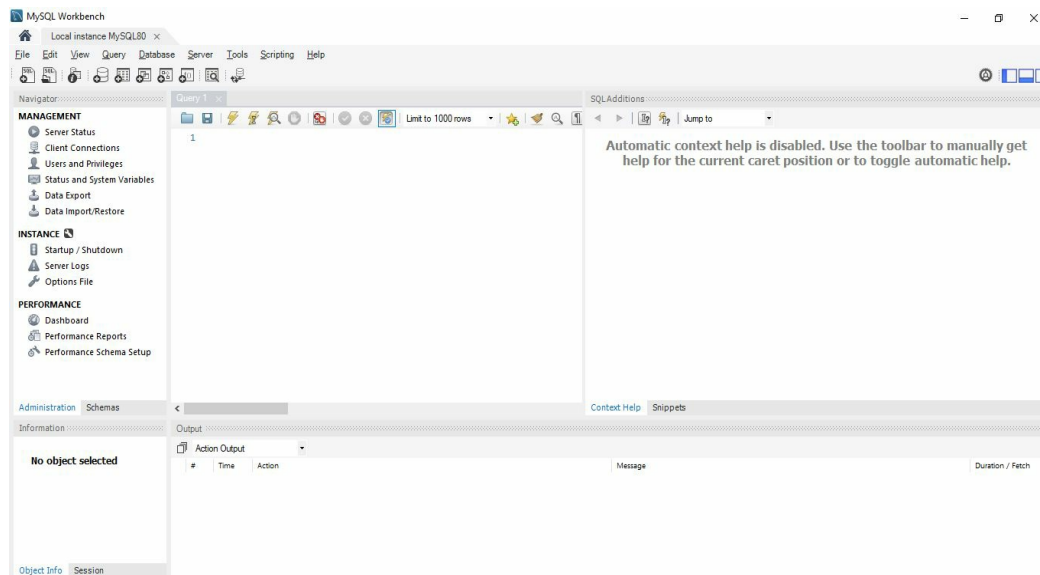
## Getting Started With SQL

For this tutorial, the exercises have been done on a computer running the Windows Operating system. As you are by now aware, there are many versions of SQL Server. For this book, we will be using MySQL and running most of the statements through MySQL Workbench.

Open MySQL Workbench.

Click on the MySQL Connection to access the database.

Your screen should look similar to the example below.



## MySQL Screen

On the top left-hand corner of the screen is the “Home” icon which is

represented by a house. If you click on the “House” icon it will take you back to the Welcome screen.

Next to the “House” icon, there is the “Local instance MySQL80” tab. This is the instance you are currently working in.

Below these tabs are the menu and quick access ribbon that will change to reflect the chosen menu option.

Below the menu ribbon on the left is the Navigation panel.

Below the menu ribbon in the middle is the Query Editor where you will be writing your queries.

Below the menu ribbon on the far right is the SQL additions screen.

At the bottom of the screen on the left, there is the Administration tab. This is for the administration of the database(s). You select this tab to start and stop the server, run various health checks, etc.

Schemas list the current databases that are running in the currently selected MySQL server instance. If you had to take a look at the Schemas now you would see the “sys” database. This is the internal database of the system. You may have two other test databases on the system, namely, “Sakila” and “World”.

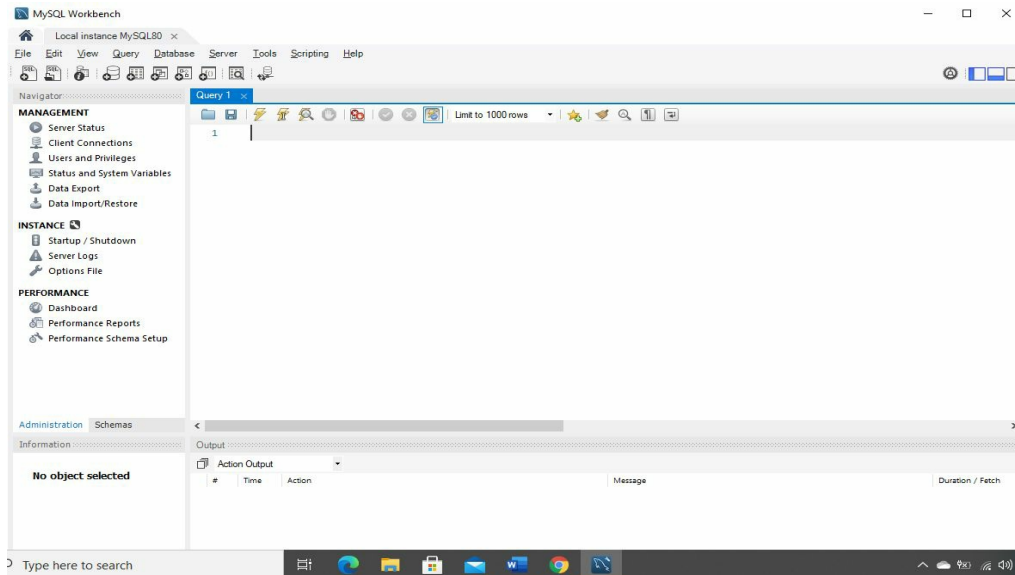
The Query tab or Instance tab is where you will be writing and executing queries.

## **Working With MySQL Databases**

### **Creating a Database with MySQL Workbench**

On the top right-hand corner of the screen, you will see three boxes each with a portion of the box colored in. These boxes hide various panels on the screen. This is handy to keep the screen clean and not confuse you.

Hide the bottom and right-hand panel on the screen. This will make the screen easier to use while you navigate SQL.

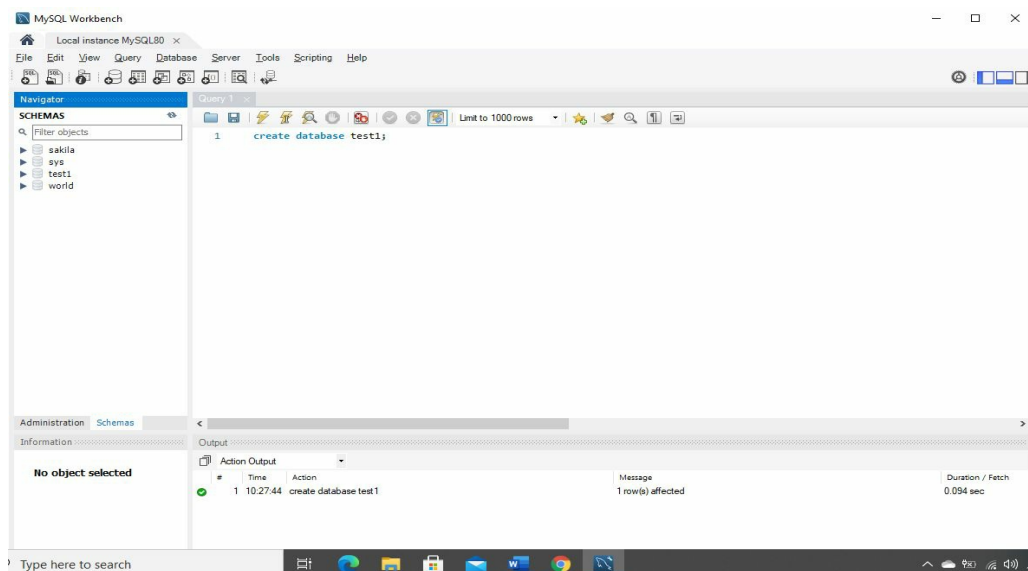


To create a database, start in the “Queries” tab and type the following:

```
create database test1;
```

This will create a database called test1. Each statement in SQL needs to end with or be separated by a semicolon (;). This allows for multiple statements to be executed within the same call to the SQL Server.

Click on the lightning bolt icon above the statements. The lightning bolt is what executes a selected script, or if there is no particular script selected it will execute all the statements.



Make sure you have the “Schemas” tab selected.

Below the statement portion of the screen, you will find the “Output” section. This shows whether or not the execution of the statement was successful.

Refresh the “Schemas” tab and you should now see the test1 database listed.

## **Deleting a Database with MySQL Workbench**

If you want to delete a database in MySQL, click on a new row in the “Queries” tab and type the following:

```
drop database test1;
```

Highlight the statement and then click on the Lightning bolt icon above the statement window. When you have a statement selected, when you click on the execute icon only that statement will be executed.

## **Working With Tables**

In the chapter above you were introduced to the concept of tables. In this section, you are going to work with basic table concepts.

Before moving on to the next section, create a database called School1.

If you have more than one database you will need to set the database you are working on as the default schema.

Highlight the database and right-click.

Choose “Set as Default Schema”.

School1 should now be highlighted.

## **Types of Storage Engines Supported by SQL**

When you create a database you want it to be optimized for the best performance, reliability, and stability. The handling of various SQL operations is done by an underlying SQL software component called a storage engine.

Usually, the default method is sufficient but at times, depending on the type of data and tables, you could run into performance issues.

There are a few different storage engine types each with their pros and cons. Some of these storage engines to consider are:

## **Archive**

Archive is best for the storage of historical data that is seldom accessed or referenced. Archive is not designed for everyday database access.

- The tables are not indexed.
- Compression is done upon insert.
- Has no transaction support.
- It is best for the archiving and retrieval of historical archived data.

## **Blackhole**

This engine is best in a testing environment where there is no need to store data.

- Blackhole will accept data.
- Blackhole does not store data.
- Useful if data is stored in a different location to the local database (distributed database environment).
- Useful for testing environments.
- Has a lot of similarities to UNIX/ dev/null.

## **CSV**

If data needs to be sent to other applications such as Excel, the CSV storage engine stores data tables in a comma-delimited format text file(s). It is important to note that CSV tables do not get indexed. A way to get around this is to initially have an InnoDB storage engine and switch to CSV when you are ready to send the files to the program in CSV format.

- An excellent engine for sharing files to programs that are compatible with CSV format.
- CSV file formats are not indexed in SQL.

## **InnoDB**

This is the default storage engine for MySQL and will create InnoDB tables as the default. InnoDB offers the following values to a database:

- It offers row-level locking.
- It offers integrity constraints for FOREIGN KEY.
- With the use of non-locking reads, multi-user concurrency is

increased.

- It is fully compliant with ACID.
- It has commit, crash-recovery, and rollback.
- It gives overall excellent performance for nearly every type of database and can be used for the majority of applications.

## **MyISAM**

ISAM has been around for a long time and is known for its speed.

- MyISAM is best suited for Data warehousing applications.
- MyISAM has no transaction support.
- It does offer full-text search indexes.
- MyISAM offers table-level locking.
- MyISAM performs best on read-heavy applications.
- MyISAM is not ACID compliant.

## **NDB**

- NDB is also known as NDBCLUSTER.
- NDB is used for clustered environments.
- NDB is best suited for distributed computing environments.
- NDB operates well where an environment requires high-redundancy.
- NDB offers one of the highest uptimes and high availability.

## **Creating a Table**

The first table to be created for the School1 database will be the Student table.

Type the following into the Query tab for the School1 database using MySQL Workbench.

```
create table Students(  
id int not null auto_increment,  
sname varchar(25),  
fname varchar not null (35),  
address varchar (45),  
marks int,  
primary key (id)  
);
```



Highlight the entire statement and execute it.

Refresh the Schema and you will notice that Tables will now show the newly created “Students” table.

## Adding Data to a Table

In this section, you are going to insert some data into the table so you have some information stored in the database.

Ensure you have the School1 database selected. You are going to be using the “**INSERT**”, “**INTO**”, and “**VALUES**” statements to insert data into the table. Starting in a new row in the Query tab type the following:

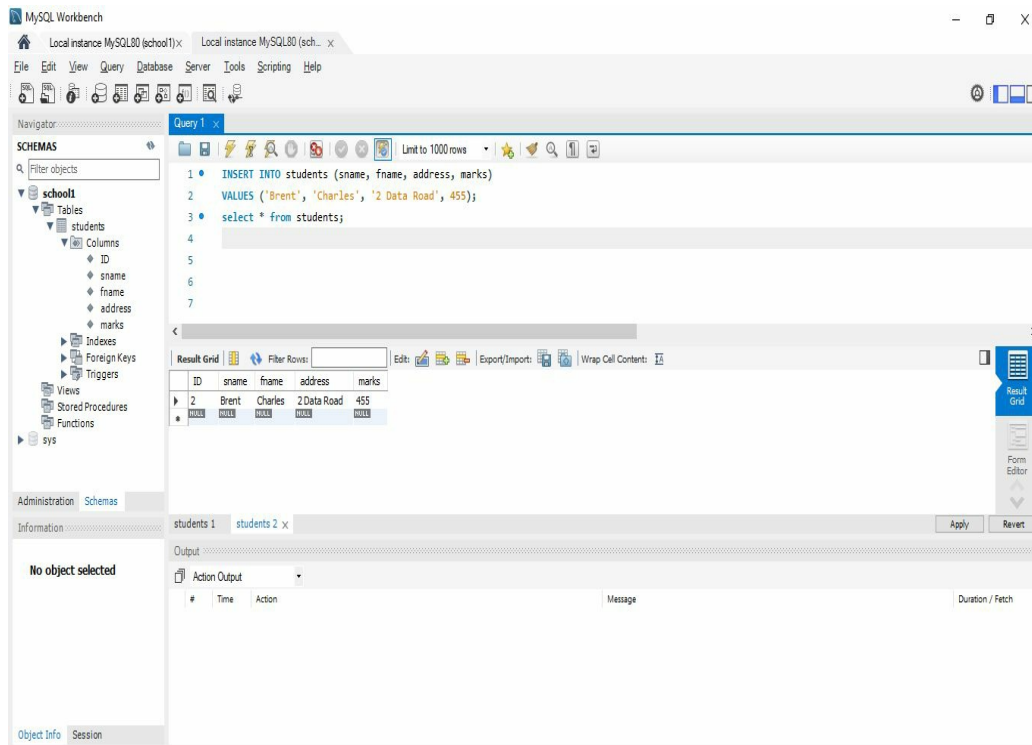
```
insert into students (sname, fname, address, marks)
values ('brent', 'charles', '2 data road', 455);
```

Highlight both lines and execute the statement.

To see if the data has been inserted into the table, use the “**SELECT**” query by typing in the following in a blank Query tab line:

```
select * from students;
```

You should see a table similar to the example below showing the data you inserted into the table.



You will notice the green checkmarks in the Output section which shows that the statement ran through correctly and without error.

You will also notice that the ID column automatically added an ID number. This is because it was set to auto\_increment when you set up the ID column. You can leave off auto\_increment if you need to insert your ID into the field.

## Exercise 1

Using the statements you have learned, complete the following tasks:

Delete the **School1** database.

```
drop database school1;
```

Create a database called **Shop1**

```
create database Shop1;
```

Set the **Shop1** database as the **default** schema.

Create the following **tables** and corresponding **fields**:

**Customers**

**Fields**

- Cust\_ID — auto\_increment NOT NULL
- Sname — varchar (35)
- Fname — varchar (35)
- Address — varchar (50)
- Phone — varchar (20)

## MySQL Statement

```
create table Customers(
cust_id int not null auto_increment,
sname varchar (35),
fname varchar not null (35),
address varchar (50),
phone varchar (20),
primary key (Cust_ID)
);
```

## Data to Populate the Fields

```
insert into customers (sname, fname, address, phone)
values ('barnes', 'jackie', '2 jupiter rd', '555-6565656');
insert into customers (sname, fname, address, phone)
values ('carter', 'john', '5 milky way', '555-7777777');
insert into customers (sname, fname, address, phone)
values ('pearce', 'sam', '15 uphill way', '555-8689894');
```

## Products

### Fields

- Prod\_ID — auto\_increment NOT NULL
- Product — varchar(45)
- Descript — varchar(50)
- Price — decimal (19, 2)

## MySQL Statement

```
create table products(
prod_id int not null auto_increment,
product varchar(35),
descript varchar not null (50),
unit int,
price decimal (19,2),
primary key (prod_id)
);
```

## Data to Populate the Fields

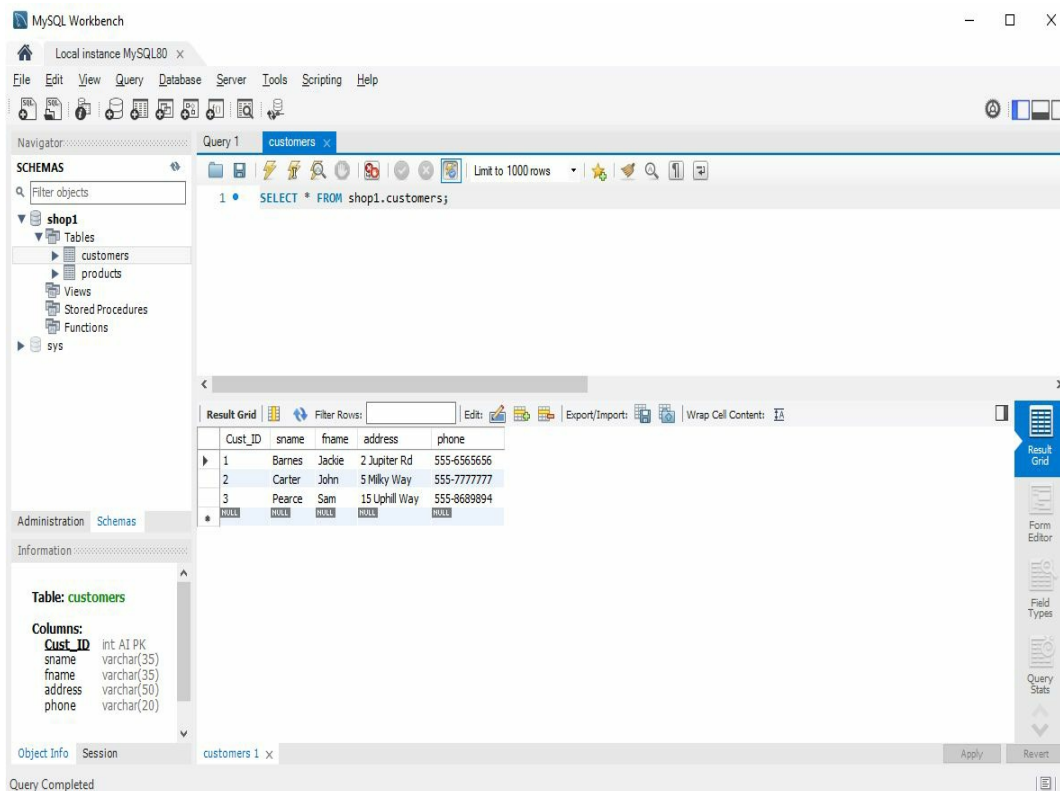
```

insert into products (product, descript, unit, price)
values ('apples', 'red and green apples', 1, 1.20);
insert into products (product, descript, unit, price)
values ('pears', 'green pears', 1, 1.25);
insert into products (product, descript, unit, price)
values ('oranges', 'large oranges', 1, 1.30);
insert into products (product, descript, unit, price)
values ('bananas', 'imported bananas', 4, 3.10);

```

## Viewing Data in a Table

To view data in an individual table you can use the navigation bar on the left by selecting the table you want to view. When the table is selected you will notice that if you hover over the table, three icons appear to the right of it. Click on the tiny grid with a lightning bolt on it and the data the table has been populated with will appear in the “Results grid”. It should look similar to the image below:



You will also notice that you have been switched to a new tab with the name of the table on it and the select statement appears in the Query screen.

# CHAPTER 4:

## Data Types

Different types of data are captured, and each type can be used to manipulate the captured data in certain ways. A store would have data relating to its products such as the types of products, the products, the amount they have in stock, and the price. If you were going to put the store's data into a database you would have three different data type fields, namely:

- Product — This would be a character data type
- Amount of stock on hand — This would be an integer data type
- Price of the product — This would be a currency or decimal data type

There are many different data types and each one has a specific purpose. The more advanced you get with database development, the more advanced working with database types become.

## SQL Data Type Categories

SQL data types can be broken into different data type categories, and these categories and the data types they represent are:

### Approximate Numeric Data Types

SQL defines numbers to be either approximate or exact. Approximate numeric data types are floating-point numeric data types.

The data types for this category are:

**FLOAT**

Starting at - 1.79E + 308 and ending at 1.79E + 308

## **REAL**

Starting at - 1.79E + 308 and ending at 1.79E + 308

## Binary Data Types

A binary data type includes images, PDF files, video files, Word files, Excel files, etc. They can be set to store either a variable-length or fixed-length field to store these types of files.

The data types for this category are:

## **BINARY**

This data type has a maximum length of 8 000 bytes, fixed length.

## **IMAGE**

This data type has a maximum length of 2 147 483 647 bytes, variable length.

## **VARBINARY**

This data type has a maximum length of 8 000 bytes, variable length.

## Character Strings Data Types

Character string data is used for normal text or alphanumeric text fields.

The data types for this category are:

## **CHAR**

This data type can store a mix of most of the data types such as numeric data that is not used in a calculation, text, and alphanumeric text. This data type has a maximum length of 8 000 characters but is a fixed-length data type.

## **VARCHAR**

This data type can store a mix of most of the data types such as numeric data that is not used in a calculation, text, and alphanumeric text. This data type has a maximum length of 8 000 characters but is variable-length data.

## Date and Time Data Types

If you have worked in any type of database or spreadsheet program you will know that there is a certain format for inserting the date and time. You can

use a character string field, but that will not enable you to do date-based calculations. It is always best to set a date and time field up as such.

The data types for this category are:

### **DATE**

The way the date is stored can be set by the regional settings on the system the database is running on. This data type will store the date in the standard format, ex. May 02, 2020.

### **DATETIME**

This will store both the date and time with a date range from Jan 1, 1753, to Dec 31, 9999.

### **SMALLDATETIME**

This data type has a smaller date range window starting from Jan 1, 1900, to Jun 6, 2079.

### **TIME**

This data type will store the time and can be set through the regional settings on the system to display the time in a customized format. An example of how it stores the time is: 13:00.

### **Exact Numeric Data Types**

Exact numeric data types are data types where the scale and the precision of the data value have to be preserved.

The data types for this category are:

### **BIGINT**

This data type takes 8 bytes of storage and represents 64-bit integers. The data type can range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 for signed values, while unsigned values have a minimum value of 0 and a maximum value of  $2^{64}-1$ .

### **BIT**

This data type is used to store bit values which can range from 1 to 64.

### **DECIMAL**

Decimal is used to define data that requires exact numeric data such as money. You have to define the number of digits and decimals, for example *decimals* (6, 2) which will show as 000000.00. The values for this field range from -999.99 to 999.99. If a calculation returns more decimal points than have been specified, the decimal will be automatically rounded up to the nearest ten.

## **INT**

This data type stores integer values, and it uses 4 bytes of storage. It has a minimum value of -2147483648 for signed values (data that can be a negative value), and a maximum value of 2147483647 for signed values. Unsigned values have a minimum value of 0 and a maximum value of 4294967295.

## **MEDIUMINT**

This data type stores integer values, and it uses 3 bytes of storage. It has a minimum value of -8388608 for signed values (data that can be a negative value) and a maximum value of 8388607 for signed values. Unsigned values have a minimum value of 0 and a maximum value of 25516777215.

## **NUMERIC**

The same rules that apply to decimal apply to numeric, as this data type is used in the same way as the decimal data type is.

## **SMALLINT**

This data type stores integer values. It uses 2 bytes of storage. It has a minimum value of -32768 for signed values (data that can be a negative value) and a maximum value of 32767 for signed values. Unsigned values have a minimum value of 0 and a maximum value of 65535.

## **TINYINT**

This data type stores integer values, uses 1 byte of storage, has a minimum value of -128 for signed values (data that can be a negative value) and a maximum value of 127 for signed values. Unsigned values have a minimum value of 0 and a maximum value of 255.



## Default Data Type Values

A default value can be a literal constant or expression in SQL. Literal constants are known either as literals or constants and are values that cannot be altered during statement execution.

The example below creates a date column that takes the current date and subtracts 20 years:

```
date default (current_date - interval 20 year)
```

# CHAPTER 5:

## SQL Statements and Clauses

One of the first things you need to understand when setting up an SQL database is the difference between an SQL Statement and SQL Clause.

### SQL Statements

An SQL statement is the query or action part of the SQL command. For instance:

```
drop table school1;
```

The SQL statement in the above example is “**DROP TABLE**” which will delete an entire table from the database.

The following are the most commonly used SQL statements:

#### **ALTER DATABASE**

This statement is used to add, delete, modify, or update file groups, files, and records associated with a database.

#### **ALTER TABLE**

This statement is used to add, delete, modify, or update tables in a database.

#### **CREATE INDEX**

This statement is used to fine-tune the database by creating an Index that allows for faster retrieval of the data in a database.

#### **CREATE TABLE**

This statement is used to create a new table in a database.

### **CREATE TABLE AS**

This statement is similar to using the “File Save As” command in Word or Excel. It allows you to create a new table using the structure and data of an existing table.

### **DELETE**

This statement is used to delete data/records from a table.

### **DROP DATABASE**

This statement is used to delete an entire database from the SQL Server.

### **DROP INDEX**

This statement is used to delete an index in a database.

### **DROP TABLE**

This statement is used to delete an entire table from a database.

### **INSERT INTO**

This statement is used to insert data into a table.

### **SELECT**

This statement is used to retrieve certain or all records from a table.

### **SELECT LIMIT**

This statement is used to retrieve a limited number of records from a table.

### **SELECT TOP**

This statement is used similar to the SELECT LIMIT statement and will return only the top records in a given query.

### **TRUNCATE TABLE**

This statement is used to delete all the records from a table without any rollback.

### **UPDATE**

This statement is used to update the records in a table.

# SQL Clauses

Some statements will be followed by a clause or condition of a query, for example:

```
select price from products where x = 2.30;
```

In the above example, the query is selecting a price from the product table that must equal 2.30. This example has the “**SELECT**” statement followed by the “**FROM**” and “**WHERE**” clauses/conditions.

The following are the most commonly used SQL clauses:

## **DISTINCT**

This clause is used to retrieve distinctive/unique records from a table.

## **FROM**

This clause is used to combine information or list information in tables.

## **GROUP BY**

This clause is used to group information by one or more columns in a table.

## **HAVING**

This clause is used to only return certain groups or rows in a query.

## **ORDER BY**

This clause is used to sort information in a query, much like the sort by command in Word, Excel, etc.

## **WHERE**

This clause is used to filter results by given criteria.

# CHAPTER 6:

## SQL Expressions, Functions, and Operators

To work with a database and the tables it houses, you need a way to manipulate the data to return the output required. You create a database to not only store, view, and retrieve the data, but to be able to manipulate the data to make it useful.

Take the “Shop1” database you created in an earlier chapter of this book. You created two tables, a customer table to hold customer records and a product table to hold product information. These tables can do more than just collect information about customers and products. The data can be manipulated to create even more useful information such as orders, invoices, and so on.

Expressions, functions, operators, and conditions allow you to perform specific tasks to create various outcomes.

### Expressions

Expressions are usually called value expressions in SQL. An expression can be complex or extremely simple. It can contain statements, values, columns, operators, etc. It is even possible to combine multiple expressions into one expression, as long as the outcome of the expression reduces to a single value. An expression can be likened to a formula that is written as a query and as such can be used to query tables for specific sets of data.

There are five different kinds of expressions in SQL:

## Conditional Value Expressions

A conditional value expression is one of the more complex types of expressions as the value of the expression depends on a condition. Although they are for advanced SQL, you should be aware of the following conditional expressions:

**CASE**

**NULLIF**

**COALESCE**

## Datetime Value Expressions

The DateTime value expressions will always return date and time as these types of expressions are used to manipulate and perform date and time operations.

The following are common expressions of this type:

**DATE**

**INTERVAL**

**TIME**

**TIMESTAMP**

To write an expression that returns data for a month from today, it would look similar to the following:

```
current_data + interval '30' day
```

## Interval Value Expressions

The interval value expression is used for scenarios such as subtracting one date from another date to get an interval time. For instance, subtracting your age from the current year will give you your birth year. The expression would look similar to the following:

```
date (current_date - interval 20 year) month to year
```

## Numeric Value Expressions

Numeric value expressions require the data to be numeric values of which

arithmetic operators can be used to formulate the result.

For example:

```
12 * 24  
-109  
25/5
```

The field name can be specified as long as the outcome equates to a numeric value, for example:

```
price * qty  
price/5  
Oranges + Apples + Pears
```

## String Value Expressions

String value expressions are used to concatenate expression values. The concatenate expression is referenced by two vertical lines — ||

For example, if you had the following expressions:

```
'grey'  
'goose'
```

You can concatenate them into a single expression:

```
'grey' || ' ' || 'goose'
```

Instead of two separate expressions the above will return:

```
'grey goose'
```

## Operators

An SQL operator is a reserved character, arithmetic operator, or reserved word that can be used to perform various arithmetic or comparison functions in a statement. There are different types of SQL operators, namely:

### Arithmetic

#### Addition

The “+” symbol is used for addition.

Example:

```
If variable a = 5 and variable b = 10  
a + b
```

will return:  
15

## Division

The “/” is used for division.

Example:

If variable a = 25 and variable b = 5  
 $a / b$   
will return:  
5

## Modulus

The “%” symbol is used for modulus.

Example:

If variable a = 100 and variable b = 10  
 $b \% a$   
will return:  
10

## Multiplication

The “\*” symbol is used for multiplication.

Example:

If variable a = 10 and variable b = 8  
 $a * b$   
will return:  
80

## Subtraction

The “-” symbol is used for subtraction.

Example:

If variable a = 50 and variable b = 15  
 $a - b$   
will return:  
35

## Comparison

### Greater Than



The “>” symbol is used for greater than.

Example:

If variable a = 25 and variable b = 5  
( $a > b$ )  
will return:  
*True*

## Less Than

The “<” symbol is used for less than.

Example:

If variable a = 25 and variable b = 5  
( $a < b$ )  
will return:  
*False*

## Equal

The “=” symbol is used for equal to.

Example:

If variable a = 25 and variable b = 25  
( $a = b$ )  
will return:  
*True*

## Not Equal

The “!=” or “<>” symbols are both used to represent not equal to.

Example:

If variable a = 25 and variable b = 5  
( $a != b$ )  
will return:  
*True*  
or  
( $a <> b$ )  
will return:  
*True*

## Greater Than or Equal

The “>=” symbols are used for greater than or equal to.

Example:

If variable a = 25 and variable b = 5  
(a >= b)  
will return:  
*True*

## Less Than or Equal

The “<=” symbols are used for less than or equal to.

Example:

If variable a = 25 and variable b = 5  
(a <= b)  
will return:  
*False*

## Not Greater Than

The “!>” symbols are used for not greater than.

Example:

If variable a = 25 and variable b = 5  
(b !> a)  
will return:  
*True*

## Not Less Than or Equal

The “!<” symbols are used for not less than or equal to.

Example:

If variable a = 25 and variable b = 5  
(a !< b)  
will return:  
*True*

## Logical

### ALL

Compares all values in one data set to a specified value.

### AND

This operator is used with the WHERE clause to allow for multiple conditions in a statement.

## **ANY**

Compares a value to similar values in a referenced list.

## **BETWEEN**

This operator is used to return values that are between a minimum and maximum value.

## **EXISTS**

This operator is used to search for an existing row in a table that has been specified that meets a specific criterion.

## **IN**

This operator compares specified literal values in a given list.

## **LIKE**

This operator uses wildcard operators to compare similar values in a table or list.

## **NOT**

This operator negates the logical operator used alongside it. For example:

not in not between not unique not exist
--

## **OR**

This operator is used to string together multiple conditions when used with the WHERE clause in a statement. The value can be x or it can be y, for instance.

## **IS NULL**

This operator compares NULL values with a value.

## **UNIQUE**

This operator is used to search for duplicates in a table.

## **Functions**

SQL has a set of built-in functions and can facilitate user-defined functions. Functions are used to manipulate and process data and are used throughout an SQL database.

Some of the more commonly used built-in functions in SQL are:

### **AVG**

This function returns the average of an expression.

### **CONCAT**

This function will concat string values.

### **COUNT**

This function will count the number of rows in a database.

### **MAX**

This function will return the maximum value of a given range of values.

### **MIN**

This function will return the minimum value of a given range of values.

### **RAND**

This function will generate a random number.

### **SQRT**

This function will return the square root of a specified value.

### **SUM**

This function will return the sum of a given range of values

# CHAPTER 7:

## Working With Constraints

A constraint is a rule that is placed on an entire table, column, or selection of columns. It restricts what types of data may be added to the table or column.

To ensure the integrity of the data being stored in a cell, table, or database, the developer needs to assign certain constraints to it. There are two levels of constraints in SQL and these are:

- **Column level** constraints which will only affect a single column.
- **Table level** constraints which will affect the entire table.

Constraints can be set when the table is first created or after the table has been created.

## Commonly Used SQL Constraints

### CHECK

This constraint ensures that the data that is imputed into a cell/column meet and satisfy the criteria that have been set for that column.

An example of using the CHECK constraint would be in a database that has an age group selection. For a health and fitness database, you may want to set different exercise regimes per age group.

The CHECK constraint can be used to limit a column's value between two age groups such as 35 to 45.

Example:

```
create table exercises (
```

```
cust_id int not null auto_increment primary key,  
name varchar (45) not null,  
age int not null check (age >=35 and age<= 45)  
);
```

The above example:

- Creates a table called exercises.
- It populates the table with the following fields:
  - **cust\_id** — This is the primary key, the number is automatically inserted upon creation of a record. It will be created in the form of an integer data type. The field cannot be blank.
  - **name** — This is a variable character field that is set to forty-five characters long, the field cannot be blank.
  - **age** — This field is an integer field that cannot be blank and will only accept values from 35 up to and including 45.

## DEFAULT

This constraint is used to insert or specify the default value(s) for a column. The value is automatically inserted into the column by the database engine.

Example:

```
create table product (  
prod_id int not null auto_increment primary key,  
product varchar (30) not null,  
price decimal (3,2) not null,  
qty int (5) not null default '4'  
);
```

The above example:

- Creates a table called product.
- It populates the table with the following fields:
  - **prod\_id** — This is the primary key, the number is automatically inserted upon the creation of a record. It will be created in the form of an integer data type. The field cannot be blank.
  - **product** — This is a variable character field that cannot be blank.
  - **price** — This field is a decimal field set to 3 digit and 2 decimal places. The field cannot be blank.

- **qty** — This field is an integer field that will assign the default quantity of the product. For the particular example, each product comes in a quantity of four which will automatically be assigned upon creation of a new record in the table.

## FOREIGN KEY

This is a unique identifier that is used to identify a related record in a different table.

Example:

In Chapter 3 of this book under the “Table Exercise” section, you created the “shop1” database. You also created two tables for the shop1 database:

- Customers
- Products

Each of the above tables had a unique primary key:

- Customers table primary key = cust\_id
- Products table primary key = prod\_id

If you were going to have an ordering system you would need to create another table. For this example, the new table will be called orders.

For the ordering system to work, the orders table needs to pull information from the customers and the products table. The way the system does this is by referencing the primary keys from both tables which become known as a FOREIGN key to the orders table.

Take a look at the code below:

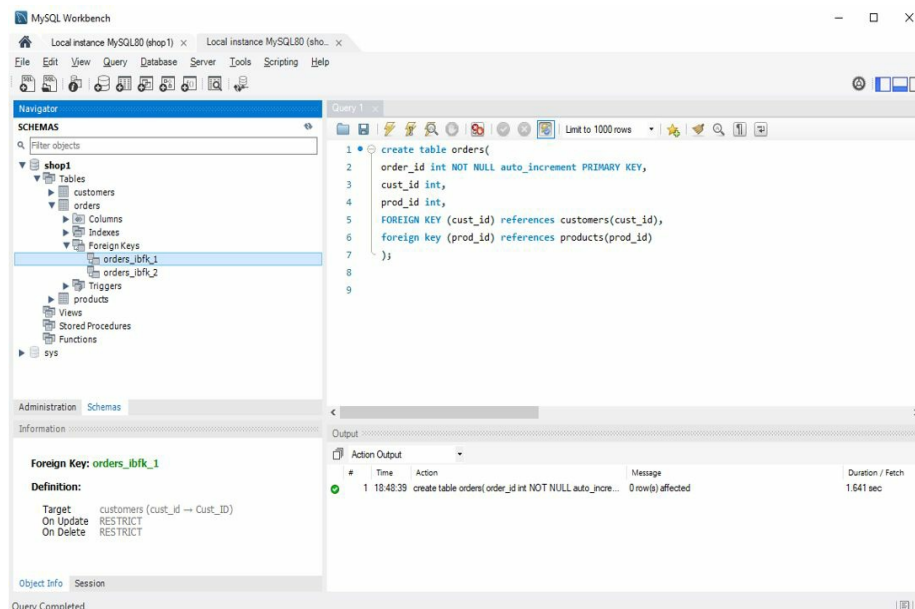
```
create table orders (  
  order_id int not null auto_increment primary key,  
  cust_id int,  
  prod_id int,  
  foreign key (cust_id) references customers (cust_id),  
  foreign key (prod_id) references products (prod_id)  
);
```

The above example:

- Creates a table called orders.
- Populates the table with the following fields:

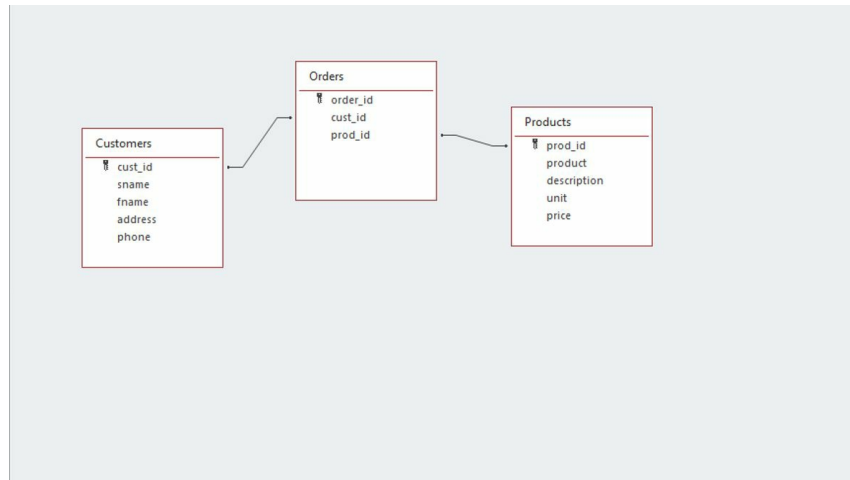
- **order\_id** — This is the primary key, the number is automatically inserted upon the creation of a record. There can only ever be one primary key in a table.
- **cust\_id** — This is the column that references the cust\_id in the customer's table.
- **prod\_id** — This is the column that references the prod\_id in the products table.
- **FOREIGN KEY (cust\_id) REFERENCES customers (cust\_id)** — This sets the first foreign key in the order table to reference the customer's table to link to the cust\_id field in the table.
- **FOREIGN KEY (prod\_id) REFERENCES products (prod\_id)** — This sets the first foreign key in the order table to reference the products table to link to the prod\_id field in the table.

There can be more than one foreign key in a table. In MySQL Workbench the newly created order table will reflect two foreign keys and look similar to the image below.



The relationship table will look similar to the diagram below.





## NOT NULL

When you see a NULL value in a table it means that the cell/record/column is blank. In other words, it does not contain any values or data, it has a NULL value. The NOT NULL constraint is used to ensure that the cell is not blank. It will create an error if no data is entered into it.

## PRIMARY KEY

The primary key is the unique ID that is given to each record in every row of every table in a database. You can set the primary key to auto\_increment which will allow the database engine to automatically generate the ID. You can also set it to be anything you want it to be and manually enter it.

For instance, your customer ID may be alphanumeric, ABC001. Always make sure the primary key field is set to NOT NULL to ensure a value is placed in the field.

There can only be one primary key column per table.

## Adding Constraints

To add a constraint during the initial setup of a table the syntax would be similar to:

```
id int not null auto_increment,  
primary key (id)
```

## Altering Constraints

To alter an existing constraint in a table the syntax would be similar to:

```
Alter table students drop constraint id primary key;
```

## Exercise 2

Before continuing on to the next chapter you are going to create a new database called “hr” using some of the information you have learned so far.

### The HR Database

**Starting on a new query line, type the following:**

```
create database hr;
```

Execute the command.

**Starting on a new query line type the following:**

```
create table salaries (  
  sal_id int not null auto_increment,  
  jobtitle varchar (45),  
  joblevel varchar (10),  
  salary int not null,  
  primary key (sal_id)  
);
```

Execute the command lines.

**Starting on a new query line type the following:**

```
create table departments (  
  dept_id int not null auto_increment,  
  dept varchar (40) not null,  
  primary key (dept_id)  
);
```

Execute the command lines.

**Starting on a new query line type the following:**

```
create table employees (  
  emp_id int not null auto_increment,  
  name varchar (55) not null,  
  age int not null,  
  primary key (emp_id)  
);
```

Execute the command lines.

**Starting on a new query line type the following:**

```

insert into departments (department) values ('IT Department');
insert into hr.departments (department) values ('Accounting Department');
insert into hr.departments (department) values ('Helpdesk Department');
insert into hr.departments (department) values ('HR Department');
insert into hr.departments (department) values ('Snr Management');
insert into hr.employees (name, age) values ('John Green', '35');
insert into hr.employees (name, age) values ('Sandy Smith', '28');
insert into hr.employees (name, age) values ('Terry Peach', '31');
insert into hr.employees (name, age) values ('Jack Sunny', '47');
insert into hr.employees (name, age) values ('Dru Red', '51');
insert into hr.employees (name, age) values ('Jane Web', '30');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Administration', 'Entry 1', '25000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Clerk 1', 'Entry 2', '28000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Clerk 2', 'Mid 1', '31000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Management', 'Mid 2', '35000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Engineer 1', 'Entry 3', '26000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Engineer 2', 'Mid 2', '32000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Engineer 3', 'Mid 3', '40000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Sn Manager 1', 'Snr 1', '43000');
insert into hr.salaries (jobtitle, joblevel, salary) values ('Director', 'Snr 2', '55000');

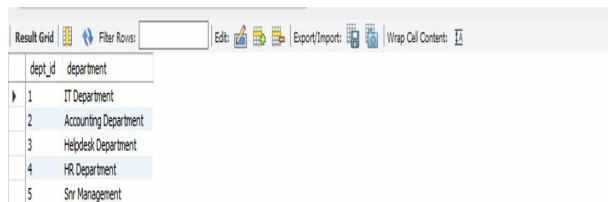
```

Execute all the “insert into” command lines

**Starting a new query tab type the following:**

```
select * from hr.departments
```

Execute the command and the “Results Grid” should look similar to the image below



dept_id	department
1	IT Department
2	Accounting Department
3	Helpdesk Department
4	HR Department
5	Snr Management

**Starting a new query tab type the following:**

```
select * from hr.employees
```

Execute the command and the “Results Grid” should look similar to the image below

Result Grid			Filter Rows: <input type="text"/>	Edit:			Export/Import:		Wrap Cell Content:
	emp_id	name	age						
▶	1	John Green	35						
	2	Sandy Smith	28						
	3	Terry Peach	31						
	4	Jack Sunny	47						
	5	Dru Red	51						

**Starting a new query tab type the following:**

```
select * from hr.salaries
```

Execute the command and the “Results Grid” should look similar to the image below

Result Grid			Filter Rows: <input type="text"/>	Edit:			Export/Import:		Wrap Cell Content:
	sal_id	jobtitle	joblevel	salary					
▶	1	Administration	Entry 1	25000					
	2	Clerk 1	Entry 2	28000					
	3	Clerk 2	Mid 1	31000					
	4	Management	Mid 2	35000					
	5	Engineer 1	Entry 3	26000					

# CHAPTER 8:

## Joins, Unions, Ordering, Grouping, and Alias

In a database, you are going to want to combine the information from one or more tables into one table. You can do this using *Join* and *Union*, but it is important not to get the two confused. In this chapter, you are going to learn the basics of *join* and *union*, and how they work.

### JOINS

Consider the *shop1 database* created in an earlier chapter. There is the *products table* and the *customers table*. These tables collect valuable information about the store's products and customers. In Chapter 7 you created the *orders table* which references the *products table* and *customers table*. The information stored in the *orders table* does not show much information about either table except for ID codes.

Before you start working with joins, populate the *orders table* with some example data.

In a new “Query” line, insert the following data into the *orders table*:

```
insert into shop1.orders (cust_id, prod_id) values ('1', '2');
insert into shop1.orders (cust_id, prod_id) values ('3', '2');
insert into shop1.orders (cust_id, prod_id) values ('3', '4');
insert into shop1.orders (cust_id, prod_id) values ('2', '1');
insert into shop1.orders (cust_id, prod_id) values ('2', '3');
```

Currently the *orders table* returns information similar to the table below:

	order_id	cust_id	prod_id
1	1	2	
2	3	2	
3	3	4	
4	2	1	
5	2	3	
*	NULL	NULL	NULL

The *orders table* would not make much sense to someone who did not know all the customer and product identification numbers. When dealing with a large customer and product databases with long lists of customers and products, only seeing the codes is not going to be useful.

The information that gets filtered into the database can be selected by joining tables. For this exercise use the *shop1 database* to *join* information between the *orders table* and the *customers table*.

Using MySQL Workbench, set the *shop1 database* as the *default* database by typing the following into a new “Query” line:

```
select order_id, sname, fname
from orders
join customers on orders.cust_id=customers.cust_id
```

Execute the script and the “Results Grid” should look similar to the one in the image below.

	order_id	sname	fname
1	1	Barnes	Jackie
4	4	Carter	John
5	5	Carter	John
2	2	Pearce	Sam
3	3	Pearce	Sam

The *orders table* now reflects the customer’s details and not just the *cust\_id*.

Notice that there is no *cust\_id* field. This field may be required, especially for reporting purposes.

If the *cust\_id* field had been included in the following script, the script would not have run:

```
select order_id, cust_id, sname, fname
from orders
join customers on orders.cust_id=customers.cust_id
```

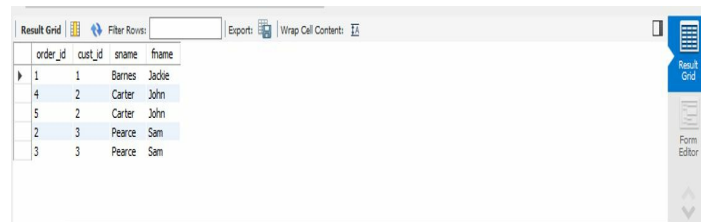
Instead, it would have thrown out an error similar to the example below:

“Error code: 1052 Column ‘emp\_id’ in field list is ambiguous”

This error occurs because the `cust_id` field appears in both tables being joined. To get around this error and show the `cust_id` type the code as follows:

```
select order_id, orders.cust_id, sname, fname
  from orders
 join customers on orders.cust_id=customers.cust_id
```

The `cust_id` field will now appear in the query.

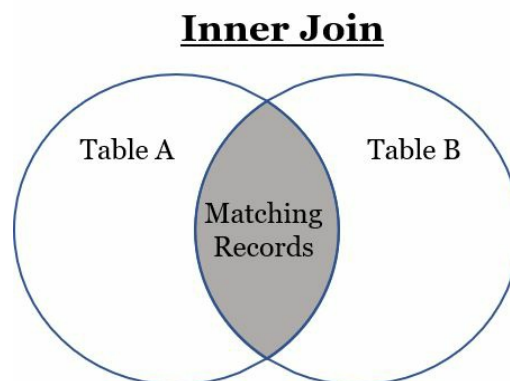


order_id	cust_id	sname	fname
1	1	Barnes	Jackie
4	2	Carter	John
5	2	Carter	John
2	3	Pearce	Sam
3	3	Pearce	Sam

## Types of Joins

There are different types of SQL joins, and each will return a different result when used in a table.

### INNER JOIN



*Join* and *inner join* are the same function. If you are working in a database with different types of joins, using inner join makes the statement a lot clearer.

The *inner join* or *join* will only return records that are a match in both *Table A* and *Table B*.

### Example:

Using the hr database created in Exercise 2, type the following in a new

“Query” line:

```
create table workforce (  
  wf_id int not null auto_increment primary key,  
  emp_id int,  
  dept_id int,  
  sal_id int,  
  foreign key (emp_id) references employees (emp_id),  
  foreign key (dept_id) references departments (dept_id),  
  foreign key (sal_id) references salaries (sal_id)  
);
```

Execute all the command lines.

Starting on a new “Query” line, type the following:

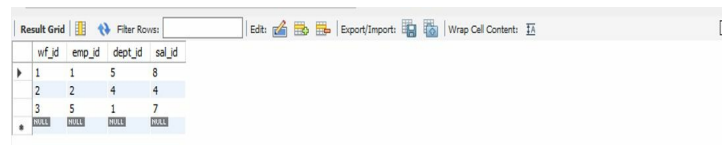
```
insert into workforce (emp_id, dept_id, sal_id) values ('1', '5', '8');  
insert into workforce (emp_id, dept_id, sal_id) values ('2', '4', '4');  
insert into workforce (emp_id, dept_id, sal_id) values ('5', '1', '7');
```

Execute all the command lines.

Run the following query:

```
select * from hr.workforce
```

Execute the command line and the results will be similar to the following image:




	wf_id	emp_id	dept_id	sal_id
1	1	5	8	
2	2	4	4	
3	5	1	7	
4	NULL	NULL	NULL	NULL

Using an *inner join* to populate the *workforce* table with the *emp\_id*, *name*, and *age* from the *employees* table, type the following code in a new “Query” line:

```
select wf_id, employees.emp_id, name, age, dept_id, sal_id  
  from workforce  
  inner join employees on workforce.emp_id=employees.emp_id
```

The “Results Grid” should show the *workforce* table as below:



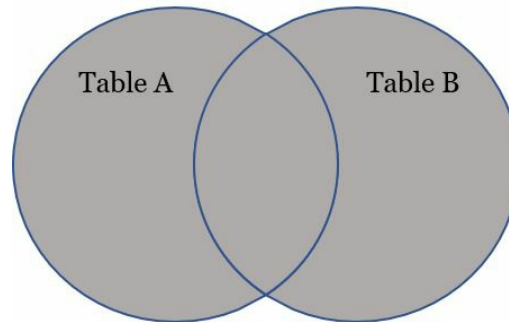
	wf_id	emp_id	name	age	dept_id	sal_id
1	1	John Green	35	5	8	
2	2	Sandy Smith	28	4	4	
3	5	Dru Red	51	1	7	



The *inner join* returns only the records from the *workforce table* that match the records from the *employees table*.

## FULL JOIN (OUTER)

### Full Join



The *full join* is also known as the *outer* or *full outer join*. It will show all the records in both *Table A* and *Table B*, regardless of whether they match or not.

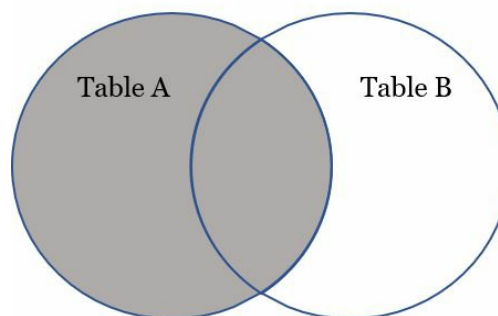
To use the *full join* to populate the *workforce table* with the *emp\_id*, *name*, and *age* fields from the *employees table*, the code should look similar to the following:

```
select wf_id, employees.emp_id, name, age, dept_id, sal_id
  from workforce
 full join employees
```

The *join* will return all the records in the *workforce table* and the *employees table*.

## LEFT JOIN

### Left Join



The *left join* will show all the records in *Table A* regardless of whether they match the records in *Table B*.

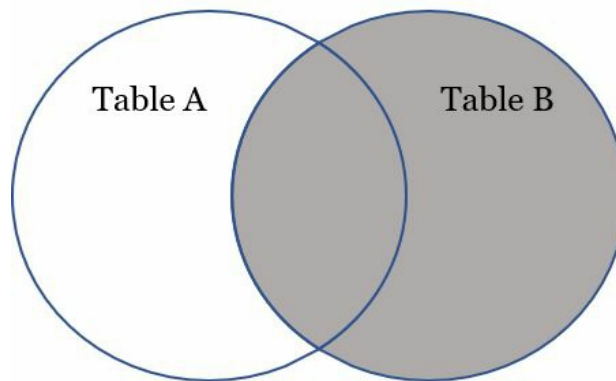
To use the *left join* to populate the *workforce table* with the *emp\_id*, *name*, and *age* fields from the *employees table*, the code should look similar to the following:

```
select wf_id, employees.emp_id, name, age, dept_id, sal_id
from workforce
left join employees on employees.emp_id=workforce.emp_id
```

The *join* should have returned all the records from the *workforce table* and only the matching records from the *employees table*.

## RIGHT JOIN

### Right Join



The *right join* will show all the records in *Table B* regardless of whether they match the records in *Table A*.

To use the *right join* to populate the *workforce table* with the *emp\_id*, *name*, and *age* fields from the *employees table*, the code should look similar to the following:

```
select wf_id, employees.emp_id, name, age, dept_id, sal_id
from workforce
left join employees on employees.emp_id=workforce.emp_id
```

The *join* returns all the records from the *employees table* and only the matching records from the *workforce table*.

## Joining More Than Two Tables

Consider the *hr database* for instance. It is not only the *employee table* information that is required to make the *workforce table* usable. What about the information from the *departments table* and the *salaries table*?

In the examples above only the *dept\_id* and *sal\_id* information is reflected in the *workforce table* query. More than two tables can be joined together using the various SQL joins.

To use the *right join*, to populate the *workforce table* with the *emp\_id*, *name*, and *age* fields from the *employees table*, the code should look similar to the following:

```
select wf_id, employees.emp_id, name, age, departments.dept_id, department, salaries.sal_id,
jobtitle, joblevel, salary
from workforce
inner join employees on employees.emp_id=workforce.emp_id
inner join departments on departments.dept_id=workforce.dept_id
inner join salaries on salaries.sal_id=workforce.sal_id
```

Execute all the command lines.

The *join* returns all the records from the matching records from the *employees table*, *departments table*, and the *salaries table* that match the records listed in the *workforce table*.

Try using each of the different types of joins to join the tables in the *hr database*.

## UNION

It is easy to get confused with *join* and *union* as they are both used to join information. The difference between these two types of joins is that *join* is used to filter and combine data from two or more different tables. The *union* clause is used to join the results of two or more queries without returning any duplicate information.

Example:

If certain information was required from the *employees table* such as a criterion for a certain age group, the *select* statement would be similar to:

```
select age from employees where age >= 30 and age <=40;
```

Type the query above into a new “Query” line and execute all the command lines.

The “Results Grid” will list all the employees in the *employees table* that are

within the set query age parameters.

From the workforce table, information needs to be extracted to find what jobs earn between \$25,000 and \$35,000. The *select* statement would be similar to:

```
select salary from salaries where salary >= 21000 and salary <=32000;
```

Type the query above into a new “Query” line and execute all the command lines.

The “Results Grid” will list all the job titles in the *salaries table* that are within the set query salary parameters.

If you had to create a *union* between the two queries above:

```
select age from employees where age >= 30 and age <=40;  
union  
select salary from salaries where salary >= 21000 and salary <=32000;
```

The above union would combine the two queries with the information specified in both select statements. However, the above example is a basic example of what the *union* clause does.

There are a few rules that apply when using the *union* clause:

- The data types must be the same in each *select* statement.
- The data must follow the same order in each *select statement*.
- There must be the same number of selected columns in each select statement.
- There must be the same number of column expressions in each select statement.
- There does not need to be the same amount of records in each select statement.

Example:

Create a small customer and order database as follows:

```
create database orders1
```

Set the *orders1 database* as the default schema in MySQL Workbench.

Create tables for the *orders1 database*.

In a new “Query” line type the following:

```
create table customer (  
  cust_id int not null auto_increment,  
  custname varchar (55),  
  address varchar (55),  
  olimit dec (10. 2),  
  primary key (cust_id)  
);
```

Execute the script.

In a new “Query” line type the following:

```
insert into customer (custname, address, olimit) values ('Henery', 'address 1', '2300.34');  
insert into customer (custname, address, olimit) values ('Janet', 'address 2', '2000.00');  
insert into customer (custname, address, olimit) values ('John', 'address 3', '4500.50');  
insert into customer (custname, address, olimit) values ('Jessie', 'address 3', '5200.00');  
insert into customer (custname, address, olimit) values ('Daisy', 'address 4', '3500.00');
```

Execute the script.

In a new “Query” line type the following:

```
create table orders (  
  ord_id int not null auto_increment primary key,  
  cust_id int,  
  oddate date,  
  amount decimal (10,3),  
  ostatus varchar (25),  
  foreign key (cust_id) references customer (cust_id)  
);
```

Execute the script.

In a new “Query” line type the following:

```
insert into orders (cust_id, oddate, amount, ostatus) values ('1', '2020-03-12', '100.50','Pending');  
insert into orders (cust_id, oddate, amount, ostatus) values ('3', '2020-07-10', '1000.00','Shipped');  
insert into orders (cust_id, oddate, amount, ostatus) values ('4', '2020-07-15', '500.50','Complete');  
insert into orders (cust_id, oddate, amount, ostatus) values ('4', '2020-09-19', '2300.00','Processing');
```

Execute the script.

In a new “Query” line type the following query:

```
select customer.cust_id, custname, address, oddate, amount from customer  
left join orders on customer.cust_id=orders.cust_id
```

Execute the script and take note of the results the query returned.

In a new “Query” line type the following query:

```
select customer.cust_id, custname, address, oddate, amount from customer
right join orders on customer.cust_id=orders.cust_id
```

Execute the script and take note of the results the query returned.

Now create a union between the two queries. In a new “Query” line type the following query:

```
select customer.cust_id, custname, address, oddate, amount from customer
left join orders on customer.cust_id=orders.cust_id
union
select customer.cust_id, custname, address, oddate, amount from customer
right join orders on customer.cust_id=orders.cust_id
```

Execute the script and take note of the results the query returned. You will notice that there is now no repeated information in the table.

The more advanced you become in SQL, the more you will be able to refine the *union* clause.

## Order

The *order* clause is used much like the “Sort By” command in programs such as MS Word, MS Excel, and various databases. The *order* clause is used to sort data in the tables by selected columns in ascending or descending order.

Using the *customer table* in the *order1 database* run the following scripts:

```
select * from customer
order by custname
```

Execute the script.

You will notice that the table has been sorted alphabetically from a to z (ascending) order.

Using the *customer table* in the *order1 database* run the following scripts:

```
select * from customer
order by custname, olimit
```

Execute the script.

You can sort data in a table by more than one column.

Try the following script:

```
select * from customer
```

```
order by olimit, custname
```

Execute the script.

You select the first column the data is to be sorted by, by referencing that column name first.

You can sort the table in descending order:

Try the following script:

```
select * from customer  
order by custname desc;
```

Execute the script.

Now the order of the records runs from z to a. The same rules that apply to sort the columns in ascending order (default) apply to sort the columns in descending order.

## Group By

The *group by* clause is used to group identical data. For instance, if you look at the *orders table* in the *order1 database*, some customers have placed more than one order.

To get the total amount a customer has spent on orders, the *group by* clause can be used.

Try the following example to give you a better idea of how this clause works.

```
select ord_id, customer.cust_id, custname, sum(amount) from orders
join customer on customer.cust_id = orders.cust_id
group by cust_id;
```

Execute the script.

The results of the above script will show all the customers who have placed orders and total the amount of their orders.

## Alias

There are times when you need to shorten the name of a table or column. The following script is a prime example. You can shorten the fields highlighted in bold below to make working with the table a lot simpler.

```
select customer.cust_id, customer.custname, orders.amount
join customer on customer.cust_id = orders.cust_id
```

Using an alias for the script above would reduce the syntax to the following:

```
select ord_id cust.cust_id, custname, sum(amount)
from customer as cust, orders as o
where cust.cust_id = o.cust_id;
```



# CHAPTER 9:

## Stored Procedures

As you have been working through the various chapters of this book you have been writing quite a few scripts. You may have noticed some of the scripts have recurring code and some scripts are exactly the same as other scripts that you had written before.

When you are writing quick small scripts as you have been doing in the exercises above, it is not that time-consuming. But what if you had complex queries that performed various tasks on a weekly or daily basis? Some complex scripts can carry on for many lines. When you are writing scripts such as those, it is very easy to miss the odd comma, underscore, brackets, and so on. Once the script is perfect, you are going to want to have a way to store the script so you can use it again and again.

### **The Advantages of Using Stored Procedures**

There are a few benefits to using SQL stored procedures. Some of these benefits are:

- Stored procedures increase the execution speed of the script. Execution is faster because stored procedures are compiled, optimized, parsed, and stored in the database cache.
- Executing a stored procedure is less bandwidth-hungry if you are using a network because the procedure is executed from the local database cache.
- Stored procedures can be used over and over again without having to rewrite complex scripts.

- Stored procedures offer great security and have tighter restrictions and constraint options.

## Creating Stored Procedures

Using the following script you used in the *group by* section of the previous chapter, you are going to create a stored procedure.

In a new “Query” line type the following:

```
select ord_id, customer.cust_id, custname, sum(amount) from orders
join customer on customer.cust_id = orders.cust_id
group by cust_id
order by custname;
```

If you were using the *orders database* in a real-life situation, the order table would be large and would more than likely grow per day. The above script is one that would most likely be used for an invoicing procedure.

On a new “Query” line create a stored procedure as follows:

```
delimiter $$
use `orders1` $$
create procedure `totalcustorder`()
begin
select ord_id, customer.cust_id, custname, sum(amount) from orders
join customer on customer.cust_id = orders.cust_id
group by cust_id
order by custname;
end; $$
delimiter ;
```

Execute the script.

In the “Navigation Panel” of MySQL, under the “Schemas Tab”, refresh the “Schemas”.

Expand the *orders database*. Below the database, there are the “Tables”, “View”, and “Stored Procedures”.

Expand the “Stored Procedures”. You will see the newly created *totalcustorder* procedure.

## Executing Stored Procedures

There are two ways of executing stored procedures.

## MySQL “Navigator Panel”

Expand the “Stored Procedures” in the navigations panel.

Hover over the *stored procedure* that is to be executed. Click on the lightning bolt icon that appears upon hover.

### Use a Call Function

You can write a small script:

```
call orders1.totalcustorder();
```

# CHAPTER 10:

## **Views, Index, Truncate, Top, Wildcards, and Triggers**

You can use views as virtual tables that do not hold any data with their contents being defined by a query. The advantage of using a view is that it adds a layer of security to the database by restricting access to certain columns or rows.

A view can also restrict data being viewed to a summary view instead of a detailed one. They can be used to protect the data layer while still allowing viewing access to it.

### **Encrypting a View**

You can create a view without columns that contain sensitive data, thus hiding the data you don't want to share. You can also encrypt the view definition, which returns data of a privileged nature. Not only are you restricting certain columns in a view, but you are also restricting who has access to the view. However, once you encrypt a view, it is difficult to get back to the original view detail. The best precaution is to make a backup of the original view.

### **Creating a View**

To create a view in MySQL, expand the database you want to create the view for.

Right-click on Views.

Select New View.

The View Designer will appear.

Type the following in the script that is pre-written on the screen using the *shop1* database:

```
CREATE VIEW `customerview` AS select cust_id, sname, phone  
from shop1;
```

Execute the script by clicking on the “Apply” button on the bottom right-hand side of the “Query” screen.

Refresh the “Schemas” navigation panel in MySQL.

Expand the *shop1* database.

Expand the *Views* beneath the *shop1* database.

Hover the mouse over the *customerview* you created.

Click on the small table with a lightning bolt icon and the “Result Grid” will bring up the view.

You can order columns for views and create views that pivot tables (see the chapter in this book on how to Pivot tables in MySQL). Creating views in SQL allows you to limit what certain users can and cannot see right down to the data records in a table.

## Index

Indexing data searches speeds up the data lookup and access to the tables/records that have been indexed. They work much like the Index or TOC in a book. Instead of flipping through hundreds or thousands of pages looking for information, you can refer to the book’s index to find the page you are looking for. This cuts down the time and effort of having to scan through all those pages.

In a database, creating an index will speed up a search using the select query and the where clause. But, you have to be careful with indexes because they may speed up lookup but they can have a negative impact on updating and inserting information into a table. But creating or dropping indexes will not affect the data in tables and is relatively easy to do.

There are different types of indexes that you can create within a database and these are:

## Table Indexes

This creates an index for an entire table.

To create an index on a table you will use the *create index* command and the statement will look similar to the following statement:

```
create index testindex on customers;
```

This will create an index for the entire *customers* table in the *shop1* database.

## Single-Column Indexes

You can create an index for a single column within a database table by typing in a statement such as the one below:

```
create index testindex2 on customers (sname);
```

This will create an index for the *sname* column in the *customers* table of the *shop1* database.

## Composite Index

You can also create an index that will index more than one column within a table. This is called a composite index and the statement to create this index is as follow:

```
create index testindex3 on customers (sname, fname);
```

The above statement will index both the *sname* and the *fname* columns of the *customer* table in the *shopt1* database.

## Implicit Index

The moment you set up a table in a database it will automatically create an index for the new object. Unique constraints and primary keys will automatically be indexed by the SQL engine as an implicit index.

## Unique Index

Unique indexes test for the integrity of data and do not allow any duplicate entries to be entered into a table. A unique index can be created by using the

following SQL statement:

```
create unique index testindex4 on products (products);
```

This will create a unique index on the products table and ensure there are no duplicate product names in the products table of the shop1 database.

## Deleting an Index

You can delete an index when it is no longer useful or is taking up too many resources. To do this, you use the *drop index* statement. This can be done by using the following statement:

```
alter table customers  
drop index testindex3;
```

## Indexing Tips

There are times when using an index is necessary and can greatly enhance the performance of a search through millions of database records. But there are also times when indexes should not be used.

Here are a few tips on when to use or not use database indexes:

- Indexes are necessary when searching through large amounts of data.
- Indexes are not necessary when the database tables are small.
- Avoid using indexes on data that is not searched frequently.
- If columns in a table contain NULL values it is not a good idea to use indexes.
- Tables and columns that are used for large batch inputs, updates, and modifications should not be indexed.

## Truncate

The truncate command is used to completely delete all the data from a table. This differs from the *drop table* command in that *truncate* only deletes the data and not the table itself.

This is useful when you have obsolete data in a table but want to keep the structure of the table, or you have archived the current data in a table and need to use it for more recent data.

The syntax to truncate data from a table is as follows:

```
truncate table customers;
```

You can check that all the records have been removed from the table by running the following:

```
select * from customers;
```

## Top

SQL uses the *top* command to return the top *n* amount of records from a table. For example, say you had the following products with their sales data for the year:

Product	Yearly Sales
Apples	40000
Bananas	22000
Grapes	35000
Pears	48000
Watermelon	55000

You could write a query that will return the top products in the products table. To do this you would use the following statement in SQL:

```
select top 2 * from products;
```

The result of the query would return:

Product	Yearly Sales



Apples	40000
Bananas	22000

MySQL does not support the *top* command. Instead, you will need to use the *limit* statement in a similar query to the one below:

```
select * from products limit 3;
```

The above statement will return the same result as using the *top* statement in SQL does.

## Wildcards

SQL supports the use of wildcards to make selecting certain data easier. For instance, when you add the “\*” in the select statement you are using a wildcard, as the asterisks tell SQL to select “all” the columns in that particular table without having to type them out individually.

Besides the asterisk, there are two other common wildcards supported by SQL:

### The Underscore Operator “\_”

The underscore operator is used to mark place holders to create a match in a search for one character example:

```
select * from customers where fname like _ndy
```

The above statement will return all the customer names that end with “ndy” and have one character in front of the search criteria, for instance, Andy.

```
select * from customers where fname like sh____
```

The above statement will feature all the customers' names that start with “sh” and have three characters at the end of the name, for instance, Shane, Shawn, Shaye, etc.

### The Percentage Operator “%”

The percentage operator works a lot like the asterisks but is used to match one or more characters in a search:

```
select * from customers where fname like %nd%
```

The above statement will list all the customers' names that have an “nd” in them in any position of the word, such as Sandy, Andy, Mandy, etc. Some databases use the \* operator in place of the %.

```
select * from customers where fname like %am
```

The above statement will only return names that have “am” as the last two letters such as Pam, Sam, William, etc.

## Triggers

Triggers can be used to automatically execute actions within an SQL database during a DDL or DML operation.

An example of a database trigger is setting the database maintenance to run after a set period of time. Another example would be setting up notifications when certain conditions are met. Retail stores use them to run future promotions or to give certain discounts for the one-hundredth user, and so on.

SQL offers two types of triggers which are:

### AFTER Trigger

The AFTER trigger will be executed after an event or action has been performed and meets the trigger's conditions (for instance, when a certain database threshold has been exceeded, trigger an event).

### INSTEAD OF Trigger

This trigger will perform an action that is different from the actual event.

## Trigger Syntax

You can create a trigger in MySQL by expanding the Schema navigation view of the MySQL database. For this example, you can use the *shop1* database, so make sure it is set as the default schema.

Type the following in the script that is pre-written on the screen using the *shop1* database:

```
create table totalaccount (  
id int auto_increment not null,  
tamount decimal (10,2),
```

```
Primary key (id)  
);  
create trigger ins_sum before insert on totalaccount  
for each row set @sum = @sum + new.tamount;
```

Execute the script.

Refresh the “Schemas” navigation panel in MySQL.

Expand the *shop1* database.

Expand the *Triggers* beneath the *shop1* database.

You will now see the trigger you created.

This will sum the amount of the tamount column when data is entered into it.

# CHAPTER 11:

## Pivoting Tables in MySQL

Records in an SQL database are presented in rows. To *pivot* data in SQL means to rotate the data records and convert them into columns. This is handy for use in data analysis and reporting, especially for visualizing trends and multidimensional reports.

### Create a Products Sales Database

Open a new MySQL session with a new “Query” tab and type the following script:

```
create database sales
```

Set the *sales database* as the default schema in MySQL Workbench.

Create tables for the *sales database*.

In a new “Query” line type the following:

```
create table prodsales (  
  id int not null auto_increment,  
  product varchar (45),  
  syear year,  
  sales decimal (10, 2),  
  primary key (id)  
);
```

Execute the script.

In a new “Query” line type the following:

```
insert into prodsales (product, syear, sales) values ('Razor Blades', '2015', '23000');  
insert into prodsales (product, syear, sales) values ('Razor Blades','2015', '25000');
```

```

insert into prodsales (product, syear, sales) values ('Razors', '2015', '23800');
insert into prodsales (product, syear, sales) values ('Razors', '2015', '20800');
insert into prodsales (product, syear, sales) values ('Razor Blades','2016', '30000');
insert into prodsales (product, syear, sales) values ('Razor Blades','2016', '32000');
insert into prodsales (product, syear, sales) values ('Razors', '2016', '35000');
insert into prodsales (product, syear, sales) values ('Razors', '2016', '37000');
insert into prodsales (product, syear, sales) values ('Razor Blades', '2017', '43000');
insert into prodsales (product, syear, sales) values ('Razor Blades', '2017', '45000');
insert into prodsales (product, syear, sales) values ('Razors', '2017', '46000');
insert into prodsales (product, syear, sales) values ('Razors', '2017', '48000');

```

Execute the script.

Query the *prodsales* table:

```
select * from prodsales
```

Execute the script.

The *prodsales* table currently looks like the table below.

Prodsales SQL Table		
product	syear	sales
Razor Blades	2015	23000.00
Razor Blades	2015	25000.00
Razors	2015	23800.00
Razors	2015	20800.00
Razor Blades	2016	30000.00
Razor Blades	2016	32000.00

Razors	2016	35000.00
Razors	2016	37000.00
Razor Blades	2017	43000.00
Razor Blades	2017	45000.00
Razors	2017	46000.00
Razors	2017	48000.00

## How to PIVOT Data in MySQL

MySQL does not support *pivot*, nor do a few other RDBMS databases such as Microsoft SQL Server and SQLite. However, like MySQL, some functions can simulate the functionality of *pivot*.

In MySQL, the *case function* can be used to *pivot* tables. The *case function* is a conditional statement. It will check to see if a condition is met before returning a result. It uses the same type of format as the “if ... then ... else ...” and is usually followed by an aggregate function or two such as *when* and *then*.

A simple explanatory example of using the *case* statement would be to consider a simple customer address log where you want to group customers by the towns they live in.

This simple script would look something like:

```
select residents, town, state, zip
from states
order by (CASE WHEN town is NULL THEN zip ELSE state END);
```

The above script selects residents and sorts them by either their town, zip, or state.

In the next section, you are going to use the *case* function to pivot data using the *prodsales table* from the *sales* database you have created.

If you look at the data collected in the above *prodsales* table representation, you can see there are numerous ways you could manipulate and report on that data. One such report would be to view the number of sales per product per year as per the table below.

<b>Pivoted Prodsales SQL Table</b>			
<b>product</b>	<b>2015</b>	<b>2016</b>	<b>2017</b>
Razor Blades	47000.00	62000.00	88000.00
Razors	44600.00	72000.00	94000.00

You could extract the information with the following query:

```
Select product, sum(sales), syear
from prodsales
group by syear;
```

But this would still return the output view in a column view.

Select a new “Query” tab and start the following script to pivot the data to read as shown in the table above.

```
SELECT
product,
sum(case when syear = '2015' then sales else 0 end) as '2015',
sum(case when syear = '2016' then sales else 0 end) as '2016',
sum(case when syear = '2017' then sales else 0 end) as '2017'
FROM prodsales
GROUP BY product
```

Execute the script and the data will be presented with only the two products, each showing the sales per year.

As good practice and to keep your SQL skills fresh, save the pivot table script as a procedure that you can reference and use again.

# CHAPTER 12:

## Clone Tables

*Clone tables* are identical tables that you can create to perform SQL tasks. *Clone tables* have exactly the same format and content as the original table they are cloned from. Using a *clone table*, you can ensure that the new table has the exact same indexes, constraints, etc. that the original table has.

### Why Use a Clone Table?

- To create identical tables to work with for updates, fixes, etc. instead of working on live tables and data.
- To act as practice tables for beginners, so that the tables in the live databases are safe and protected.
- To feature new interfaces for new users.

### Using Clone Table

Using the *sales database*, you will clone the *prodsales* table created in the previous chapter and populate the *cloned table* with the data of the original table.

Open a new “Query” tab in MySQL. You will use the same script you used to create the original *prodsales table*. Type the following into a new “Query” line:

```
create table prodsales_clone (  
id int not null auto_increment,  
product varchar (45),  
smmonth varchar (25)  
year year,  
sales decimal (10, 2),
```



```
primary key (id)  
);
```

Execute the script. You now have two identical *prodsales tables* in the *sales database* to work with.

Populate the *prodsales\_clone* table with the exact same data as the original table by typing the following into a new “Query” line in MySQL:

```
insert into prodsales_clone (product, smonth, syear, sales)  
select product, smonth, syear, sales  
from prodsales;
```

Execute the script.

Query the *prodsales\_clone table*:

```
select * from sales.prodsales_clone;
```

Execute the script and the “Result Grid” will show the cloned table populated with the exact same data as the original *prodsales* table.

# CHAPTER 13:

## Security

Databases hold all kinds of sensitive data including employee records, company blueprints, and so on. Due to the nature of company data, systems need high-end, functional, and reliable security which include controlled access to systems. Many companies today use Microsoft's Active Directory to manage users and sort them into access profiles using processes such as group policies.

Group policies are used to assign employees various permissions based on their job title, security level access, etc. by using group level, directory level, and even file-level permissions for more individualized permissions.

Active Directory is compatible with SQL but only for access control and does not provide internal security for authorization. These types of security servers are provided by the applications that run SQL, such as MySQL, SMSS, Microsoft Sequel server, and so on.

### **Components of Database Security**

There are four main components of database security and these are:

#### **Authentication**

Authentication pertains to validating a user's permission to access various systems resources, directories, files, and applications.

The most common method of authentication is simply a username and password. A username and password will verify the credentials of a person

trying to access the system.

Single sign-on is available which uses secure certificate authentication. This means that the user can only sign onto the system on a certain device that has been set up with the necessary security certificates.

## **Encryption**

Most corporations go to great lengths to ensure that system access is authenticated and appropriately authorized.

Encryption strengthens this access control by scrambling data into indecipherable symbols which are hard to interpret if not correctly authenticated.

Microsoft SQL Server uses RSA data encryption. RSA is an algorithm that uses key management and a layered hierarchical structure to encrypt data. It makes use of SSL certificate key infrastructure which uses both Public and Private Keys.

## **Authorization**

The Authorization process is what determines a systems user's authentication type, in other words, what they can and cannot access.

Once a user has logged onto the system their access policies will prevent them from being able to access the information they do not have permissions to access. If the client is dealing with information, authorization can be restricted right down to the file level. Rights can be limited to being able to modify, update, delete, create, etc.

## **Access Control**

Change tracking is used to maintain and keep a log of a user's use of the system. SQL user change tracking also makes it possible to track the activities of all authorized users.

Tracking can track the day, time, and even what was accessed per workstation or user ID.

Tracking changes helps to prevent a user with top-level access from causing damage to the system.

## **Three-Class Security Model**

SQL uses a security model consisting of three classes. All three SQL security classes interact with each and have the same basic security functions, namely:

- Principals — Principals can only access certain specified objects in the database.
- Securables — Securables are system regulated resources that run within the database.
- Permissions — Permission refers to the right to view, edit, or delete securables. This access is pre-defined.

Although this security model is a Microsoft SQL Server-based idea, other SQL Management systems such as MySQL, DB2, and MongoDB have a similar security structure.

Security modeling is a lot more widespread across the IT industry. It incorporates networks, cloud computing, and personal computers or devices right down to mobile phones.

Security covers a wide area of SQL and any system that houses sensitive data. It is not a subject that can be covered in a chapter, but rather an entire course on its own.

## **Schemas**

In relation to security, the schema defines ownership of resources in a database system. Schemas identify principles based on the client's level of authorization.

According to Microsoft, a schema is “a group of database objects under the ownership of a single individual and together they create what is known as a single namespace.”

The single namespace refers to a limitation that does not allow two tables that are contained in one schema to contain similar names.

Principals are used for either group, single login, or a single user.

Multiple users sharing one role are grouped using group policies, and all can cooperatively own a schema or many schemas.

It is possible to share or transfer a schema between principals without renaming the schema.

There are T-SQL statements for managing schema, but a majority of this work belongs to database administrators, not the principals of a database.

## Server Roles

Roles offer a database system another layer of security. This layer identifies access based on a user's systems credentials such as responsibilities and access levels.

Although there are many different kinds of roles, SQL comes with both fixed database and server roles, each of which provides implicit permissions.

SQL offers the option to customize roles such as user-defined server roles, application roles, and user-defined database roles.

A few Fixed Server Roles are:

- **sysadmin** — The role of the sysadmin user holds the highest form of access and can perform any SQL database or SQL Server action.
- **bulkadmin** — This role permits users to run the *bulk insert statement*.
- **dcreator** — This role permits users to create a new database within an SQL Server instance.
- **diskadmin** — Users allocated to this role manage SQL Server disk files.
- **processadmin** — This role allows users to terminate SQL Server processes.
- **public** — This role is used for user authentication to the SQL database.
- **securityadmin** — The securityadmin role grants users permission to assign or revoke user rights to the database in an SQL Server instance.
- **serveradmin** — The serveradmin role can start and stop the SQL Server as well as assign various configuration rights.
- **setupadmin** — Users assigned this role can use T-SQL statements to remove or add new linked servers and they must have sysadmin

rights to perform this role.

Roles play an important role in securing a database and database engine. SQL security functions are mostly built on top of server roles as the authentication upon login determines the access permissions principals have to various schemas.

Data encryption protects information from potential hackers and other unauthorized access or login attempts. It also plays a vital role in protecting company data from internal threats or for information being accessed by users that should not be accessing it.

Roles govern exactly what users can access or what tasks they can perform within a database or Database server instance. SQL security practices are to allocate a user the barest minimum rights they need to effectively perform their duties.

This stops a user from having rights to the wrong information and access to valuable resources that they are either not trained to use or may be holding an unnecessary license for.

Sysadmins are considered to be database or schema owners.

Members of an organization's IT department are usually tasked or assigned the role of database administrators. There are different levels and departments of both database administrators and database owners. The level depends on the assigned task within the organization.

### **Customized Server Roles**

Server roles can also be created and customized to suit a given purpose within an organization.

Custom server roles can be created by:

- Selecting Security objects in the SQL server management program.
- Select “New”
- Select “Role”
- The name of the role can be anything, but it is usually given a name pertaining to the role and function of the role.

- The “Explicit Permissions” needs to be set. This is usually a checkmark box.
- The permissions are then set and the role is created.

## **Logins**

The security hierarchy in an SQL database starts with the systems administrator or database owner that originally creates the database instance.

The SQL security model is based on the principles as covered earlier in this chapter:

- Principals – Have access to deny, granted, or revoke access at all levels.
- Securable – These are database/database engine objects that can be manipulated and as such need access level rights to be set.
- Permissions – Permissions are set to allow various access levels to data, database tables, and securables.

## **Groups**

Groups can be set up to contain either various roles, access permission rights to specific files, databases, tables, or database functions.

Groups make granting various access and roles to a specific group of people a lot easier. For instance, a manager of an accounting department may require access to the same roles as their teams but may also need to have access to resources. These can all be set in a group, making it easier to manage all the users’ access and for granting another user the same access requirements more efficiently.

It is usually the function of the database owner or administrator to create and manage groups.

## **Mixed Mode Authentication**

Mixed mode authentication gives an extra layer of security to authentication of a user login. Mixed mode authentication means logging into the database has two-tier authentication and passwords are stored on the database.

The “Systems Administrator” or “SA” user is enabled when mixed mode authentication is set. The SA has the highest level of access rights on the

database and this user is barely used or checked; it can become a security risk. The SA is usually the first ID targeted by hackers.

SQL best practice is to rename the SA and ensure it has a strong password that is changed regularly.

Find the name of the SA or root user by typing in the following:

```
select user, host from mysql.user;
```

This will return a table similar to the following in the “Result Grid”.

User	Host
mysql.infoschema	localhost
mysql.session	localhost
mysql.sys	localhost
root	localhost

The SA or root user can be renamed in MySQL by typing the following into a new “Query” line:

```
rename user 'root'@'localhost' to 'newroot'@'localhost'  
flush privileges;  
mysql -u newroot -p
```

To change the password of the SA or root user:

```
update mysql.user SET Password=PASSWORD('thisismynewpassword') where user='root',  
flush privileges
```

After the root password has been changed you will need to completely log out and shut MySQL down. Then log in again with the new password.

## Database Roles

Permissions managed at the server level are called server roles. Permissions



managed at the database are called database roles. Database roles can assign common permissions to principles by grouping them.

The most common database roles are:

- `db_accessadmin` — Manages user access by being able to add and remove users a database instance.
- `db_backupoperator` — This user role is used to backup the database.
- `db_datareader` — This user role is permitted to use the *select* statement in permitted database instances.
- `db_datawriter` — This user role is permitted to execute Data Manipulation Language (DML) statements on permitted database and table instances.
- `db_denydatareader` — This user role is used to deny permission to run the *select* statement throughout all database instances.
- `db_denydatawriter` — This user role is used to deny permissions to perform DML statements in database instances.
- `db_ddladmin` — This user role can *drop*, *alter*, and create objects in a database.
- `db_owner` — This is the user role that has full access to the entire database.
- `db_securityadmin` — This user role is used to set permission to the database securables.

### **Customized Database Roles**

Like server roles, database roles can also be created and customized as and when required to suit a given purpose within an organization.

Custom database roles can be created by:

- Selecting Security objects in the SQL server management program.
- Select “New”
- Select “Database Role”
- The name of the role can be anything but should be something pertaining to the role and function of the role.
- The “Explicit Permissions” needs to be set; this is usually a

checkmark box.

- The permissions are then set and the role is created.

## Encryption

Encryption can be used to enhance database security. Encryption is just another layer of security and does not replace the main security control method for an SQL database.

Encryption offers the following benefits:

- It can be easy to implement.
- It does not have an impact on any applications or scripting.
- It offers protection for system backups.
- Allows for more useful and efficient audit trails.

While encryption enhances the system's security, it can have a few negative impacts on the database system as well. Some of the negative impacts of implementing encryption are:

- Encryption can have a negative impact on the system's performance.
- There is no encryption protection at the application level which leaves data vulnerable to inside attacks.
- There are usually large amounts of encryption keys to manage that can become costly and add to an overburdened database administration team.
- Encryption has been known to be disruptive to DB functions such as indexing, merge, and search.

## Master Keys

SQL Server root encryption uses service *master keys*. When a new SQL server instance is created a new *master key* is automatically generated. A *master key* is to be used to encrypt database master keys and is linked to various server passwords.

There can only be one *master key* per server instance and the key gets stored in the *master database*.

You can generate a new *master key* should you need. The script to generate a

new SQL master key is as follows:

```
alter service master key regenerate
```

SQL best practice is to create a backup of the *master key* and to create a password for it. To backup the *master key* and set a password the script would look similar to the following:

```
backup service master key  
to file = 'c:\sqlserverkeys\servicemasterkey'  
encryption by password = 'Password1'  
to restore to a previous master key, you use the following:  
restore service master key  
from file = 'c:\sqlserverkeys\servicemasterkey'  
decryption by password = 'Password1'
```

There are database master keys that should have a strong password that is changed regularly. To create a database master key, you use the following script:

```
create master key encryption by password = 'Password1'
```

To back up the database *master key* you can use the following script:

```
backup master key to file = 'c:\sqlServerKeys\serviceMasterKey'  
encryption by password = 'Password1';
```

If you need to use the backup *master key*, you can run a restore procedure such as the following one:

```
restore master key  
from file = 'c:\sqlserverkeys\servicemasterkey'  
decryption by password = 'Password1'  
encryption by password = 'Password1'
```

## Transparent Data Encryption (TDE)

If you store the data by encryption key in the boot record of a database, you can encrypt aspects of the database such as log files and data pages. To do this, pages need to be encrypted before they are written to disk, then before they are written to memory they get decrypted. This is one of the most efficient ways of encryption as it does not put unnecessary strain on the system or cause bloat.

Because the encryption method is transparent across all systems, this encryption method does not require extra coding.

Once a database master key is created a security certificate is also created, which can only be encrypted by the master key. A database is still accessible once TDE has been implemented. But some statements are affected and will **not** be accessible. These are:

- Drop a file
- Drop a filegroup
- Drop a database
- Detach a database
- Take a database offline
- Set a database as read-only

The following script can be used to create a service master and create a server certificate using the master key.

In a new “Query” tab, type the following on a new “Query” line:

```
use master
create master key encryption by password = 'Password1'
create certificate TDECertificate with subject = 'Certificate For TDE'
```

This key can be used to encrypt the database it was created in.

You can also use the *database encryption wizard* to create a TDE certificate.

# CHAPTER 14:

## SQL Injections

Hackers are the cat burglars of the Internet, and they are always on the lookout for new targets. They find vulnerabilities in systems and exploit those vulnerabilities to gain access to a system. Once in a system, they can destroy data, steal data, and so on as they have access to credit card information, bank details, and other sensitive data.

Hackers access vulnerabilities in a SQL database by using a method called “SQL Injection”. This is the most widely used method for unauthorized users to gain access to an SQL system. It is called a code injection technique because it injects malicious code through a web page into SQL statements.

A user can inject this malicious code into the system at a login screen. The user will use an SQL statement in place of a username or userid to login with. This statement passes into the system and is run through the database, unknowingly unleashing the code.

The statement below is a common authentication statement that allows users to login to a system:

```
select * from users where username='username' and password='password'
```

There is nothing wrong with the above code and it will perform the task and return the desired results. For instance, the login screen would give you the following details to fill in:

Enter Username: **Mike**  
Enter Password: **Rainyday**

Once the user enters the above details the query run on the system looks like this:

```
select * from users where username='Mike' and password='Rainyday'
```

This simple code will check for the user name Mike. If it finds the username SQL presumes the login is successful; if the username is not found the login is unsuccessful. Here the hacker can subvert the application logic by removing the password check after the *where* clause simply by using the SQL comments symbol "--".

The unauthorized user could use the following to gain full access to the SQL database:

```
Enter Username: root --  
Enter Password:
```

The "--" in SQL is a line comment statement and when it is entered everything after "--" is ignored. Simply by adding a common administrator name and then "--" after the name, SQL was told to ignore the password and allow access. Most SQL databases have an administrator, SA, or root user with full administrative rights to the database. These are the first userids a hacker will exploit.

Another common way of gaining unauthorized access is using the concept of 1=1 is always true. If you use the same username and password script you used above and entered the following user credentials:

```
Enter Username: 'jjj or' '1=1'  
Enter Password: 'jjj or' '1=1'
```

The SQL statement would run as:

```
select * from users where username=" jjj or " '1=1' " and password=" jjj or " '1=1' "
```

The above SQL statement reflects as a valid statement in SQL as  $x=x$  is always true. Thus the above username and password will gain the unauthorized access. This access will allow them to view all the users and their passwords in the SQL user table, which in turn allows the hacker access to valid users and all their details along with it.

Due to the nature and size of data out there today, the above attack can also render the database inaccessible to its valid users with a denial of service

(DoS).

The above scenario is an attack on an SQL database table for a *select* query. Imagine what an unauthorized user can do with an *update* or *delete* query. They could inject all kinds of data into the existing data and even destroy all the records in a table.

### Preventing an SQL Injection

PHP and Perl offer a function of “escaped characters” which enable the use of special characters. MySQL uses a PHP extension to ensure SQL scripts can handle the MySQL version of the escaped characters by providing the “mysql\_real\_escape\_string()”.

This string will get the data being entered and then “escape” the character to ensure that SQL sees the data being entered as text and not as code. Using this string is outside the scope of this course and is for more advanced SQL users. Here’s an example of how the code would look:

```
if (get_magic_quotes_gpc()) {$userid = stripslashes($userid);}
$userid = mysql_real_escape_string($userid);
mysql_query(Select * from customers where name='{ $userid }' ");
```

### Hacking Scenario

If you want to find vulnerable websites, Google is one of the best hacking tools in the world to do so. With the use of the *inurl:* command it is quite simple to find vulnerable websites.

Try it. You can copy and paste one of the commands below into the Google website browsers search bar, press enter, and you will get a list of websites:

```
inurl:index.php?id=
inurl:shop.php?id=
inurl:article.php?id=
inurl:pageid=
```

You can go through each of them and find their vulnerabilities.

# CHAPTER 15:

## Fine-Tuning

Investing time and effort into learning all there is to know about a database and how it works is the best way to learn about the database and the data it houses. This is important because it adds awareness to help you fine-tune and keep the database functional, effective, and useful.

By closely examining the database you will be able to understand various data trends, find vulnerabilities, and keep the database current. To confidently and effectively manage and administer an SQL database, you need to ensure you are equipped with the right credentials and knowledge to do so.

Helpful tips for learning the ins and outs of the database:

- Work with the “Third Normal Form” (3NF) design when working with an SQL database. The 3NF is a database schema that is used to design relational databases. To avoid anomalies of data, any data duplication, and ensure the referential integrity of the data, the 3NF approach uses normalizing principles.
- Wildcards should only be used when necessary. For example, using the \* will load all the information from the chosen fields. In a small database, this is not that significant. In a large database, this could mean hours of loading time and a screen full of data to sift through.
- Only *index* tables that are frequently used. Indexes take up both memory and disk space.
- Large tables that are accessed frequently should be indexed to cut



down on complex large search time. If there are no indexes a full index is run when a table is referenced. For smaller indexes, create table indexes for specific rows.

- Be careful when using equality operators, especially when you are using real numbers, times, and dates. It is likely there will be subtle differences that are not that obvious to spot, and this can cause errors. The equality operators make it nearly impossible to get an exact data match.
- Use pattern matching sparingly.
- Make sure tables are optimized for peak performance and the structure makes sense.
- Avoid the OR operator, unless absolutely necessary, as it will not only slow down searches but also add to database security vulnerabilities. Try not to use it in extra-large databases as it adds too many layers to a comparison query.
- Run regular backups. Backups should be kept at a safe site off location and in the cloud if this facility is available.
- Run regular maintenance checks on the database and ensure that the system has enough room for the database to grow. If you run out of space you run the risk of a database crash which could cause loss of data, data corruption, and application downtime.
- Defragment the database at least once a week to ensure data integrity and that the database runs smoothly.

## SQL Tuning Tools

There are a few SQL optimization tools that can be used to help fine-tune SQL and they are broken into built-in tools and application fine-tuning tools.

SQL Built-In Fine Tuning Tools include:

- **TKProf** — This tool is used to measure database performance over a specified period of time. This time is based on how long an SQL statement takes to process.
- **EXPLAIN PLAN** — This tool is used to track the path of a statement to verify the statement's credibility and efficiency.
- **SQL\*Plus Command** — This command is used to measure the elapsed time between SQL search passes.

Some of the most popular application tools are:

- SQL Query Tuner for SQL Diagnostic Manager
- SQL Server Management Studio
- EverSQL
- SolarWinds Database Performance Analyzer

# CHAPTER 16:

## **Working With SSMS**

### **Downloading SQL Server Management Studio (SSMS)**

SQL Server Management Studio is Microsoft's interface for interacting with SQL databases. It's free and is a great tool for learning and managing database servers.

Download the latest version of SSMS from the following link:

<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15>

Once the SSMA-Setup\_ENU.exe has downloaded, double click the file in the Downloads folder.

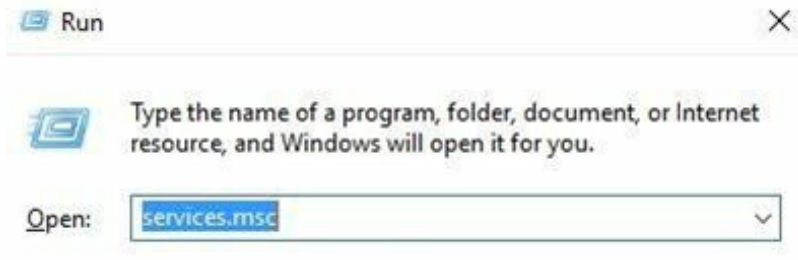
Keep the default settings and let it install.

### **Starting the Database Engine Services**

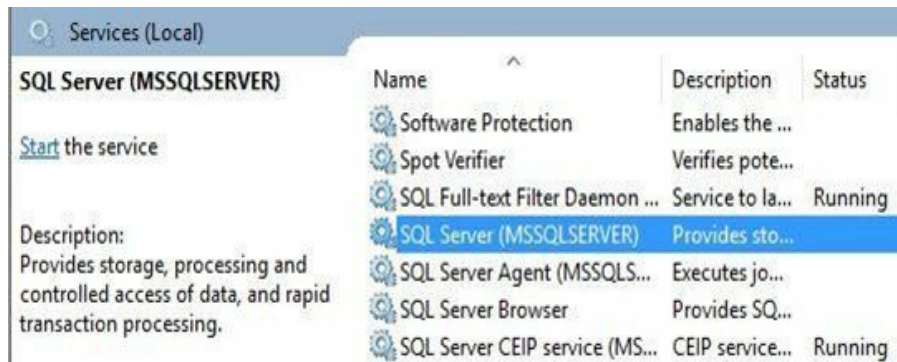
Login

Ensure that the SSMS instance is running on your system by:

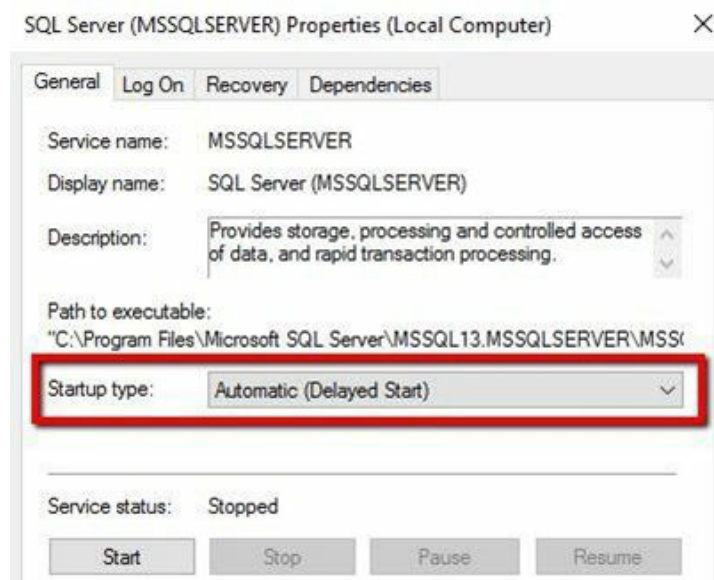
- Using the Windows key + R
- Type services.msc in the Run command box.



- This will bring up a list of services currently running on the device.
- Scroll through the services to find SQL Server.

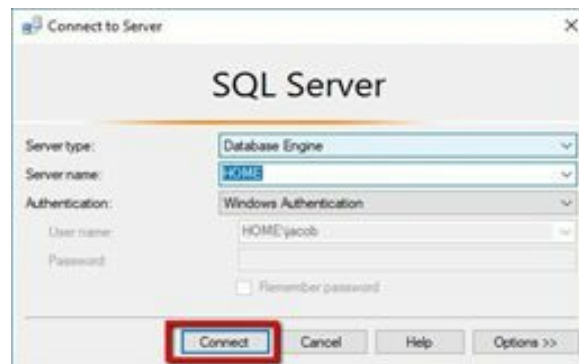


- When first installed, this service is not automatically started by default.
- Start the service if it is not currently running
- This service can consume a lot of computer resources. You can either leave it as a manual start or set it to automatically start whenever the computer is started.



## Connect to SQL Server with SSMS

- From the program, the menu finds SQL Server Management Studio.
- The following needs to be configured as follows:
  - Server Type: Database Engine
  - Server Name: (The name of your computer)
  - Authentication: Windows Authentication



You may not need to enter a password as the server will authenticate through the windows instance running on the device.

Click Connect to connect to the database engine.

## The Basics and Features of SSMS

### Managing Connections

Managing connections to servers within SSMS can be done in several ways.

The connection the red arrow points to at the top allows you access to manage the current SSMS server.

The connection the red arrow points to at the bottom gives you access to control connections to many database SSMS servers.



### Choosing Your Database

You can choose the database to manage depending on the server instance in SSMS.

You do so by choosing a database from the list of available databases listed in the drop-down menu.



## New Query Window

You will need to open a new query to run queries against the current session.

Use the “New Query” button for a new query. Alternatively, you can use the shortcut key “Ctrl + N”.

## Executing Statements

SSMS offers several easy to execute statements and they work much the same way as they do in MySQL.

- Highlight certain batches of code and execute the run command.
- If you do not select any code to execute and use the execute code command, all the code scripted on the open “Query” tab will be executed.



## IntelliSense

Intellisense is like predictive text on a mobile phone. Instead of having to type out statement names, tables, or field names, it will list what it thinks you want to use. As soon as you type the first few letters it will give you a list of options to choose from. This is handy as it cuts down on coding errors.

What is great about SSMS is that you can turn this option on and off as you need.

If for some reason, IntelliSense doesn't work as well as it should, or it stops working completely, it may mean you need to clean out the cache. To do this follow these directions:

Got to the “Edit” menu option

Choose “IntelliSense”

Choose Refresh Local Cache.

You can also use the keyboard shortcut Ctrl + Shift + R to refresh the IntelliSense cache.

## Results Presentation

The way that the result set is presented can be changed to your preference from the provided options.

The most commonly used preferences are:

- Results to Grid
- Results to Text



Keyboard shortcuts to change the presentation of the results are:

- Text results — Ctrl + T
- Grid option — Ctrl + D

Below is an example of the grid option:

Results		Messages			
Day	Exercise_Name	Rep_Goal	Actual_Reps	Weight_Used	Date_Performed
1 Monday	Bicep Curls	5	7	25.00	2017-12-05
2 Tuesday	Chest Press	5	6	35.00	2017-12-05
3 Wednesday	Squats	8	9	45.00	2017-12-05
4 Thursday	Calf Raises	6	8	30.00	2017-12-05
5 Friday	Shoulder Press	11	12	40.00	2017-12-05
6 Saturday	Back Rows	8	9	50.00	2017-12-05

Below is an example of the text option:

Results		Messages			
Day	Exercise_Name	Rep_Goal	Actual_Reps	Weight_Used	
Monday	Bicep Curls	5	7	25.00	
Tuesday	Chest Press	5	6	35.00	
Wednesday	Squats	8	9	45.00	
Thursday	Calf Raises	6	8	30.00	
Friday	Shoulder Press	11	12	40.00	
Saturday	Back Rows	8	9	50.00	
Sunday	Glute Bridge	9	12	60.00	

Ctrl + R toggles between show/hide the result set.

## Object Explorer

This is one of the most used tools in SSMS.

The object explorer interface expands and contracts the visual presentation of the navigation panel to view databases, tables, views, users, stored procedures, and more.

## Databases

The database view can be expanded to see all the objects beneath each database in a server instance. This will take you right down to the table view.

SQL Server system database summary:

- **Master** — Contains the system/server configurations. It is good practice to do regular backups of this object.
- **Model** – This is the template that is used whenever a new database is created.
- **Msdb** – This object holds the configuration for the SQL server and is used to manage the server configuration. It is used for scheduled jobs and various automated tasks.
- **Tempdb** – For the storage of temporary data files, running queries, and stored procedures. The information that exists and is used in this database is temporary and will be erased once the session ends.

There are also the ReportServer and ReportServerTempDB. These two objects represent the Reporting Services that are installed when SSMS is installed. These two services are used for:

- **ReportServer** – The main reporting server that houses custom reports, and is used to schedule jobs and notifications.
- **ReportServerTempDB** – This is a temporary reporting service that holds data in its caches until the session ends, at which time all the data is erased.



# CHAPTER 17:

## Database Administration

Databases are not only about getting set up and working. They also need to be managed, maintained, modified, fine-tuned and kept updated. This means regular integrity, performance, resilience, and various other checks.

Organizations that run and maintain an in-house database usually have a designated person or team that performs these tasks. They are a database administration (DBAT) team or database administrator (DBA).

A DBA will perform a number of different database tasks which can include the following tasks:

- **Database Integrity Checks**

Database integrity checks ensure the data structures are consistent and accurate.

- **Index Reorganization**

Updating of database tables can start to make data unravel or become messy, especially if there is indexing involved. Regular reorganizing of database indexes needs to happen to ensure the database does not become bloated, full, or lag.

- **Rebuild Index**

If an index is not functioning correctly it will need to be dropped and rebuilt.

- **Database Backup**

Backups are one of the most important parts of any system. They should be done each night. There are a few different types of backups:

- **Differential** — This will check for any changes in the data and back up those changes.
- **Incremental** — This will look for changes and only back up changes that have taken place since the last backup.
- **Full Backup** — This will backup the entire database regardless of whether or not there have been changes made.

- **Check Database Statistics**

This is to keep the newest queries being run from getting mixed up with any old queries. All old outdated queries should be deleted or archived to protect the integrity of the data.

- **Data and Log File**

Data and log files should be kept in order, with older logs that are not used either archived or deleted. The file sizes of the logs also need to be tracked as they can grow to quite a large size that takes up valuable space. Larger log files can also become hard to read and decipher. It is important to maintain smaller log files.

- **Defragmenting**

The fragmentation of the database should stay consistently below 30%. If it is higher the database will need to be defragmented. It is good practice to check on this at least two to three times a month for very large databases.

## Maintenance Plan

SQL Server Agent allows for the setting up of an automated maintenance plan. When setting up a maintenance plan, you need to get the maintenance schedule right to optimize the database and run it during off-peak usage times. This ensures that the CPU is not being over-utilized. It also does not slow it down which would impact any work being done on the database or database applications.

### Setting up a Maintenance Plan in SQL Server

The first set up establishing an SQL server plan is to identify the SQL server instances advance options. To do this you can run the following script:

```
sp_configure 'show advanced options', 1
go
reconfigure
```

```
go
sp_configure 'agent xps', 1
go
reconfigure
go
```

Once the advanced options are displayed:

- Left-click the + icon to the left of the SSMS Management screen.
- Left-click Maintenance Plans.
- Right-click Maintenance Plans
- Choose New Maintenance Plan Wizard
  - Enter the name of the maintenance plan
  - Enter an appropriate description so anyone can use it
  - From this point, you can
    - Run a specific task
    - Run all tasks
- Choose Single Schedule then click the Next button.
- Design your maintenance plan around one of the following options available within SSMS:
  - Checking your Database Integrity
  - Shrinking the Database
  - Reorganizing Index
  - Rebuilding the Index
  - Updating the Statistics
  - Clean up History
  - Executing SQL Server Agent Job
  - Backup – full, differential, or transaction log
  - Maintenance Cleanup Task
- Once you have selected the tasks this plan must perform the Maintenance Plan Wizard will guide you through setting up and fine-tuning each task.
- When you have fine-tuned the plan click Next.
- Order the tasks in order of importance. It is always best practice to set Database Backup first.
- When you have ordered the tasks click Next and you are done.

## **Defining Tasks for the Maintenance Plan**

The following is a list of handy tips for defining tasks within the Maintenance Plan for SSMS:

### **Full Database Backup Task**

This option allows you to choose which full database backup to perform when running the task in the maintenance plan. The best practice is to keep one database per plan.

The best choice for this option is to select 1.

### **Define Database Check Integrity Task**

This is an SQL Server command that runs and checks the integrity of the database, the tables, and the data populating the database.

### **Define Shrink Database Task**

You can shrink (compact) the database. This option helps to keep the database from expanding and taking up unnecessary space. It also speeds up the amount of time it takes to back up the database.

This option will let you choose which database you wish to shrink.

### **Define Reorganize Index Task**

Every time you add, modify, and delete indexes you will need to reorganize them.

The process is the same as a hard disk, where you have fragmented files and space scattered across the disk.

Perform this task at least once a week or more for a busy database.

### **Define Rebuild Index Task**

You can either reindex or reorganize the index tables in a database.

### **Define Update Statistics Task**

This option will help to keep the update statistics log current. It handles the statistics for both indexes and individual columns. You will need to select which database you want to keep the statistical logs for.

### **Define History Cleanup Task**

This is the screen that will delete or keep various historical data which can include:

- Backup information
- Recovery tables
- Agent job history
- Various other historical data

### **Define Backup Database (Differential) Task**

This option can be set to back up every page in the database that was altered since the last full backup. You will be prompted to select the database to backup.

### **Define Backup Database (Transaction Log) Task**

The transaction log backup backs up all the log records since the last backup.

### **Define Execute SQL Server Agent Job Task**

Any job that is not listed in the Maintenance Plan Wizard can be tasked here. For example:

- Check for null values.
- Check the database meets specified standards.
- Many jobs are defined here.

SQL Server Agent Job Tasks are all listed in this section of the Maintenance Plan.

### **Define Maintenance Cleanup Task**

You will define the cleanup action of the maintenance task.

This will ensure that no unnecessary files or objects are taking up disk space.

### **Report Options**

This part is to set up the Maintenance Plan reporting system. This report will send a summary of the maintenance performed and how the scheduled tasks ran.

You can either have the report saved to disk in a specific folder or have it emailed to a specific account or both.

## Running the Maintenance Plan

To run the SQL Maintenance plan you first need to check that the SQL Server Agent is running.

Press the Windows key and press the letter R.

Type services.msc in the Run box and hit Enter.

At the services screen, scroll through the services and find the SQL Server Agent (MSSQLSERVER).

Back in SSMS, right-click on the maintenance plan you created under the Maintenance Plans section.

Click Execute to start running the Maintenance Plan.

When the plan has successfully completed, click OK.

Close the dialogue box.

The maintenance reports can be viewed by right-clicking the maintenance plan you created and selecting View History. You will see all the different maintenance plans in the SQL Server instance along with the results of the plans.

To send off the database maintenance plan report you can email it.

You will first need to set up a Server Agent. To do this, right-click on SQL Server Agent.

Select New.

Select Operator.

Allocate an operatorname.

Enter in a valid email address to which you want the report sent.

Click OK and the report will be sent to the designated email.

Right-click the maintenance plan that has just been successfully run.

Select the Modify option.

The maintenance plan design screen will appear on the right-hand side.

Here you will see a graphical representation of the completed tasks.

Click on Reporting and Logging. You will find the icon for this situated on the menu bar of the design plan. This is on the left of Manage Connections.

When the Reporting and Logging screen appears, select the “Send Report to an Email Recipient”.

Select the maintenance plan operator name to send the report to.

This sets the report up to be sent to the selected recipient each time the Maintenance Plan is run.

The running and maintenance of a database is an important job. You need to ensure you have the right plan set up for the best performance and optimization of the system you're working on.

## **Backup and Recovery**

The most important task a DBA can perform is backing up the essential database in use. When you create a maintenance plan, it's important to have backup and recovery at the top of the maintenance list in case the job doesn't get fully completed.

It is important to have an understanding of the transaction log and its relevance to running a successful database system.

### **The Transaction Log**

Whenever a change is made to the database, be it a transaction or modification, it is stored in the transaction log.

The transaction log is one of the most important files to ensure the integrity of a database in an SQL Server instance. A lot of importance is set around the use of and saving of the transaction log.

Every transaction log can facilitate transaction recovery, recover all incomplete transactions, roll forward a restored file, filegroup, or page to a point of failure, replicate transactions, and facilitate disaster recovery.

### **Recovery**

Choosing a recovery option for the database is one of the first things that

needs to be done.

When the SQL Server is online, and even while users are still using the database services, all three SQL database backup options can be used.

When a backup and restore are run on the SQL Server, they are run within the confines of the SQL database recovery mode.

SQL has three database recovery model options:

### **Simple Recovery**

The simple recovery model does not make allowances for the transaction log to be backed up.

This model should only be used for databases with infrequent updates.

The log file will be truncated only allowing for minimally logged transactions.

### **Full Recovery**

The transaction log is backed up during a full recovery model. However, the log will be truncated once the backup process begins.

You can recover to any point in time with the full recovery option. In this model, you will need all the log files pertaining to the specific back to successfully restore the data.

### **Bulk-Logged Recovery**

This recovery option is only used for bulk import operations. It is used in conjunction with the full recovery model.

It must be noted that with this option the backup cannot be recovered to any point in time. Rather you can only recover to certain points in time.

The option does not fill up the transaction log and is fast.

### **Changing the Recovery Model**

The recovery model can be changed in a few ways.

- You can change the model in the SQL Server Management Studio screen by right-clicking on the database you want to change the setting for.



- Select the Properties option.
- Select Options.
- Choose the recovery model from the drop-down box.
- Reset the model to the preferred choice.

Alternatively, you can use an SQL statement query to do so. There are a few you can choose from:

### Simple Recovery Option

```
alter database sqlebook set recovery simple  
go
```

### Full Recovery Option

```
alter database sqlebook set recovery full  
go
```

### Bulk Recovery Option

```
alter database sqlebook set recovery bulk_logged  
go
```

## Backups

Database Administrators base their backup plans on the following two measures:

- **Recovery Time Objective (RTO)** — The RTO measures the recovery period time from after a disruption notification of the business process gets logged.
- **Recovery Point Objective (RPO)** —The RPO measures the elapsed time passed during a disruption but before the data size that has been lost exceeds the maximum limit of the business process.

There are three types of database backup processes to choose from in SQL and these are:

### Full Backup

The SQL Server will create a CHECKPOINT when a full database backup is set. This checkpoint is set to make sure that existing dirty pages are written to disk. For those that don't know what dirty or clean pages are:

- A dirty page is a page that has been altered or added to since it was loaded into memory. In other words, the original file that exists on the disk and the one in the memory is no longer the same.
- A clean page is a page that has been read from disk but nothing has changed. It is still the same as its counterpart on the disk it was read from.

The entire database will then be backed up along with the majority of the transaction log to keep the transactional consistency intact. Thus the database can be restored to the most recent point in time. It will also recover all the transactions up to the time the backup started.

What this means in plain English is that you can essentially only completely restore the database back to one point of time. This will be the last full backup that was taken of the database.

The reason this is not the most ideal backup option is that there is a chance of hours of data loss. For example, if the full backup ran every 24-hours and there was database corruption during the day, you would have to restore the database back to the previous night's data. This could result in the loss of an entire day's worth of work.

### **Transaction Log Backup**

The transaction log backup will only backup the data in the transaction log. Which means only the transactions that were updated or changed in any way in between backups.

Because only recently changed files are backed up, it makes this backup model less resource-hungry. Because it does not take up a lot of resources to run, nor does it have to lock files it is backing up, backups can be performed throughout the day.

### **Differential Backup**

Differential backups are backups that can run through the course of the day as they do not require a lot of resources to run. These backups will only backup a file that has changed during the day.

If there is a database corruption, using the differential model, the data can be restored to a more recent point in time. For instance, if you set the backup to run every hour, it scans for the most recently committed files and will back them up. The only data you may lose if there is a database corruption is an hour's worth (between backups).

The SQL Server keeps track of all the different pages that have been modified via flags and DIFF pages.

## **Database Backup**

### **Performing a Backup**

To back up a database in SSMS start by right-clicking the database object in SSMS.

Select Tasks.

Select Backup.

Select the type of backup to perform (full, differential, or transaction log).

Select when to run the backups.

The copy-only backup is for running a backup that will not affect the restore sequence.

### **Restoring a Database**

To restore a database in SSMS you need to start by right-clicking the database object.

Select Tasks.

Select Restore.

Select Database.

Select the database contained in the drop-down menu.

Keep the rest of the tabs populated.

If you choose the Timeline, you can see a graphical diagram of when the last backup was created. This shows how much data was lost.

There is an option of recovering up to the end of a log, or a specific date and

time.

The Backup Timeline media button is there to help you verify the integrity of the backup media using it to restore the database.

Select if you want to restore the data to the original structure or a new location.

Specify the required restore options. You can choose too:

- Overwrite the existing database
- Keep the existing database as a copy.

The recovery state either brings the database online or allows further backups to be applied.

Click OK.

The database will start the restore which could take some time depending on how large it is.

## **Attaching and Detaching Databases**

The method of attaching and detaching databases is similar to that of a backup and restore method.

Attaching and detaching a database is used for:

- Copying the .MDF file and .LDF file to a new disk or server.
- Completely taking the database offline makes sure that the database cannot be accessed by any users or applications.
- The database remains offline until it is reattached.

A database can be taken offline — this is how you detach it. Then you can reattach a database to a new destination if need be. This is very handy if for some reason the database engine needs to come down. You can run another database engine and reattach the database to the new instance.

A database has two main file groups, these files are the:

- .MDF — This file group is the database's primary data file, which holds its structure and data.
- .LDF. — This file group is where all the transactional logging activity and history is stored in.

These are the two main database files and are the files you would move to a new location once the database has been detached.

There is another file to be aware of and that is the .BAK file. This is the backup file that gets created during a database backup. You may notice that there are different versions of this file due to the various backups that have been created.

## Detaching the Database

While a database is operational it is considered to be attached.

SQL has a stored procedure for detaching a database which makes this procedure a lot easier for a DBA.

The detach procedure is held in the “master” database store.

Have a look at the complexity of the detach stored procedure by following these steps in SSMS:

- Click on the Databases folder in the navigation panel.
- Click on the System Databases beneath the database folder.
- Click on the “master” database.
- Click on the Programmability option.
- Click on Stored Procedures.
- Click on System Stored Procedures.
- Scroll down until you find sys.sp\_detach\_db.
- Right-click on this object.
- Select ‘Modify’ in SSMS.
- This will bring up the syntax for this procedure on the Query screen.
- You can execute this code to detach the database.

The SQL syntax for detaching a database is as follows:

```
use master
go
alter database databasename set single_user
with rollback immediate
go
exec master.dbo.sp_detach_db @dbname = n'databasename',
@skipchecks = 'false'
go
```

## Attaching Databases

When the database has been detached, to get it operational again it needs to be attached.

Navigate the data directory.

Find the .MDF file for the database

Connect to an instance of the SQL database.

In SSMS open a New Query option.

Type in the following SQL script to reattach the database files, note you have to reattach both the .MDF and .LDF files.

```
create database databasename on  
(filename = 'c:\sql data files\databasename.mdf'),  
(filename = 'c:\sql data files\databasename_log.ldf') for attach
```

It should be noted that you do not have to reattach the .LDF as once you start up the database a new log will be created if it cannot find the original one.

# CHAPTER 18:

## Deadlocks

In most cases, multiple users access database applications simultaneously. This will mean that multiple transactions are being executed at one time on a database instance. When a transaction executes an operation request on a database resource such as a table, that resource is locked by default. Thus no other transaction can access the locked resource.

A deadlock occurs when two or more processes access resources that are locked by another process.

If you consider a scenario:

- TransactionA performs an operation on TableA.
- TableA is then locked by TransactionA.
- TransactionB tries to execute a parallel operation on TableA after performing a task on TableB and acquiring a lock for TableB.
- While TransactionB awaits a response from TableA, TransactionA attempts a process on TableB.
- What you now have are two transactions sitting in a deadlock awaiting a response from locked resources.

Create a dummy database by executing the following script in a new “Query” tab:

```
create database dldb;  
go  
use dldb;  
create table tablea  
(
```

```

id int identity primary key,
patient_name nvarchar(50)
)
insert into tablea values ('thomas')
create table tableb
(
id int identity primary key,
patient_name nvarchar(50)
)
insert into table2 values ('helene')

```

Execute the script to create the database and table. The code you executed will also create a deadlock.

## Deadlock Analysis and Prevention

In real-world scenarios, multiple users access the database simultaneously, which often results in deadlocks.

To analyze these deadlocks, you need the proper tools to be able to find out what is causing the deadlocks so you can create a way of preventing them.

SQL Server error logs, log what transactions and resources are involved in any deadlocks that may have occurred throughout the database.

## Reading Deadlock Info via SQL Server Error Log

The SQL Server only provides minimal information about any deadlocks that may have occurred through the database. To find more detailed information about the deadlocks, you will need to do so in the SQL Server error log.

To log deadlock information to the error log you need to turn on the *trace flag 1222*.

You can turn *trace flag 1222* on, on a global level, by executing the following script in a new “Query” tab:

```
DBCC Traceon(1222, -1)
```

You can see if the *trace flag 1222* is on, by executing the following query in a new “Query” line:

```
DBCC TraceStatus(1222)
```

The above statement results in the following output:

--	--	--	--



TraceFlag	Status	Global	Session
1222	1	1	0

The “Status value 1” indicates that the flag 1222 is on.

The 1 in the Global column implies that the trace flag has been turned on globally.

Now execute the code above to insert records into each table.

First, you need to drop *tablea* and *tableb*.

Once the tables have been dropped rerun the code below:

```
create table tablea
(
id int identity primary key,
patient_name nvarchar(50)
)
insert into tablea values ('thomas')
create table tableb
(
id int identity primary key,
patient_name nvarchar(50)
)
insert into table2 values ('helene')
```

To view the SQL Server error log execute the following stored procedure:

```
executesp_readerrorlog
```

The above stored procedure will retrieve a detailed error log. A snippet of this is shown below:

458	2017-11-01 15:51:47	Parallel redo is started for database test with worker pool size
459	2017-11-01 15:51:47	Parallel redo is shutdown for database test with worker pool size
460	2017-11-01 15:51:55	deadloc list

461	2017-11-01 15:51:55	deadlock victim=process 1fcd9514ca8
462	2017-11-01 15:51:55	process list
463	2017-11-01 15:51:55	process id=process 1fcd9514ca8 taskpriority=o log...
464	2017-11-01 15:51:55	execution Stack
465	2017-11-01 15:51:55	frame procname=adhoc line=2 stmtstart=58 stmt...
466	2017-11-01 15:51:55	unknown
467	2017-11-01 15:51:55	frame procname=adhoc line=2 stmtstart=4 stmt...

The above log is just an example; errors are different depending on the database application software being used. All the information in the log about the deadlocks will start with “deadlock-list.”

Analyze the log information retrieved by the deadlock you created by rerunning the script to create *tablea* and *tableb*.

Take note that your values may differ per column, but the information should be the same as per the table below:

<b>ProcessInfo</b>	<b>Text</b>
spid13s	deadlock-list

spid13s	<b>deadlock victim=process1fcf9514ca8</b>
spid13s	process-list
spid13s	process id=process1fcf9514ca8 taskpriority=0 logused=308 wait waittime=921 ownerId=388813 transactionname=transactionBla XDES=0x1fcf8454490 lockMode=X schedulerid=3 kpid=1968 : trancount=2 lastbatchstarted=2019-05-27T15:51:54.380 lastbatc lastattention=1900-01-01T00:00:00.377 clientapp=Microsoft SQ hostname=DESKTOP-GLQ5VRA hostpid=968 loginname=DES (2) xactid=388813 currentdb=8 lockTimeout=4294967295 clien
spid13s	executionStack
spid13s	frame procname=adhoc line=2 stmtstart=58 stmtend=164 sqlhandle=0x0200000014b61731ad79b1eec6740c98aab3ab91bc
spid13s	unknown
spid13s	frame procname=adhoc line=2 stmtstart=4 stmtend=142 sqlhandle=0x0200000080129b021f70641be5a5e43a1ca1ef67e9'
spid13s	unknown
spid13s	inputbuf
spid13s	UPDATE tableA SET patient_name = 'Thomas - TransactionB'
spid13s	WHERE id = 1
spid13s	process id=process1fcf9515468 taskpriority=0 logused=308 wai waittime=4588 ownerId=388767 transactionname=transactionA XDES=0x1fcf8428490 lockMode=X schedulerid=3 kpid=11000 trancount=2 lastbatchstarted=2019-05-27T15:51:50.710 lastbatc lastattention=1900-01-01T00:00:00.710 clientapp=Microsoft SQ hostname=DESKTOP-GLQ5VRA hostpid=1140 loginname=DE

	committed (2) xactid=388767 currentdb=8 lockTimeout=429496
spid13s	executionStack
spid13s	frame procname=adhoc line=1 stmtstart=58 stmtend=164 sqlhandle=0x02000000ec86cd1dbe1cd7fc97237a12abb461f1fc2
spid13s	unknown
spid13s	frame procname=adhoc line=1 stmtend=138 sqlhandle=0x020000003a45a10eb863d6370a5f99368760983cac
spid13s	unknown
spid13s	inputbuf
spid13s	UPDATE tableB SET patient_name = 'Helene - TransactionA'
spid13s	WHERE id = 1
spid13s	resource-list
spid13s	keylockhobtid=72057594043105280 dbid=8 objectname=dldb.dbo.tableAindexname=PK__tableA__3213E83 associatedObjectId=72057594043105280
spid13s	owner-list
spid13s	owner id=process1fcf9515468 mode=X
spid13s	waiter-list
spid13s	waiter id=process1fcf9514ca8 mode=X requestType=wait
spid13s	keylockhobtid=72057594043170816 dbid=8 objectname=dldb.dbo.tableBindexname=PK__tableB__3213E83

	associatedObjectId=72057594043170816
spid13s	owner-list
spid13s	owner id=process1fcf9514ca8 mode=X
spid13s	waiter-list
spid13s	waiter id=process1fcf9515468 mode=X requestType=wait

There are three main parts logged by the SQL server in the error log which are:

### 1. **The Deadlock Victim**

One of the processes involved in a deadlock is flagged as the deadlock victim by the SQL server. In the table above, you can see the ID of the process selected as the deadlock victim. The process is process1fcf9514ca8 and has been highlighted in bold to make it easier for you to see for this example.

### 2. **Process List**

The process list references all the processes that are involved in the deadlock. For the table and the deadlock generated by the example there are two processes involved.

The processes list shows all the details of both of these processes. The process that is listed first in the process list is the one that is selected as the deadlock victim.

The process list provides a host of information about the processes such as:

- The login information of the process.
- The isolation level of the process.
- The script that the process was trying to execute.
- And much more

### 3. **Resource List**

The resource list has information pertaining to the resources involved in the deadlock event. For instance, in the example above the resources that were involved in the event were *tablea* and *tableb*.

## **Tips for Avoiding Deadlock**

- Execute transactions in a single batch and keep them short.
- Release resources automatically after a certain time period.
- Sequential resource sharing.
- Don't allow a user to interact with the application when transactions are being executed.

# CHAPTER 19:

## **Normalization of Your Data**

With SQL, normalization is the process of taking your database and breaking it up into smaller units. Developers will often use this procedure as a way to make your database easier to organize and manage. Some databases are really large and hard to handle. If you have thousands of customers, for example, your database could get quite large. This can make it difficult to get through all the information at times because there is just so much that you have to filter through to get what you want.

However, with database normalization, this is not as big of an issue. The developer will be able to split up some of the information so that the search engine is better able to go through the database without taking as long or running into as many issues. In addition, using database normalization will help to ensure the accuracy and the integrity of all the information that you place into the database.

### **How to Normalize the Database**

Now that you understand a number of the reasons for choosing to do database normalization, it is time to work on the actual process. The process of normalization basically means that you are going to decrease any of the redundancies that are inside the database (“What is Normalization? 1NF, 2NF, 3NF & BCNF with Examples”). You will be able to use this technique any time that you want to design, or even redesign, your database. Some of the tools that you need and the processes that you should learn to make this happen include:

## **Raw Databases**

Any database that hasn't gone through the process of normalization can contain multiple tables that have the same information inside of them. This redundant information can slow down the search process and can make it difficult for your database to find the information that you want. Some of the issues that you could have with a database that hasn't gone through normalization include slow queries, inefficient database updates, and poor security.

This is all because you have the same information in your database several times, and you haven't divided it into smaller pieces to make things easier in your search.

Using the process of normalization on your original database can help cut out some of the mess and make things more efficient for both you and the user to find the information needed.

## **Logical Design**

Each of the databases that you are working on needs to be created as well as designed with the end-users in mind. You can make a fantastic database, but if the users find that it is difficult to navigate, you have just wasted a lot of time in the process. Logical model, or logical design, is a process where you can arrange the data into smaller groups that are easier for the user to find and work with.

When you are creating the data groups, you must remember that they should be manageable, organized, and logical. A logical design will make it easier for you to reduce, and in some cases even eliminate, any of the data repetition that occurs in your database.

## **The Needs of the End-User**

When designing the database, it is important to keep the end-users' needs in mind. The end users are those who will be using the database that you develop, so you will need to concentrate on creating a database that is beneficial to the customer.

In general, you will want to create a database that is user friendly, and if you



can add in an intuitive interface, this can be helpful, too. Good visuals are a way to attract the customer, but you need to have excellent performance as well or the customer may get frustrated with what you are trying to sell them on the page.

When you are working on creating a new database for your business, some of the questions that you should answer to ensure that you are making the right database for your customers include:

- What kind of data will I store on here?
- How will the users be able to access the data?
- Do the users need any special privileges to access the data?
- How can the user group the data within the database?
- What connections will I make between the data pieces that I store?
- How will I ensure the integrity and accuracy of the data?

## **Data Repetition**

When creating your database, you need to ensure that the data is not repetitive. You need to work on minimizing this redundancy as much as possible. For example, if you have the customer's name in more than one table, you are wasting a lot of time and space in the process because this duplicated data is going to lead to inefficient use of your storage space.

On top of wasting the storage space, this repetitive entry will also lead to confusion. This happens when one table's data doesn't match up or correlate with others, even when the tables were created for the same object or person.

## **Normal Forms**

A normal form is a method of identifying the levels or the depth that you will require to normalize the database. In some cases, it just has to be cleaned up a little bit, but at other times there will be excessive work required to ensure that the table looks clean and organized. When you are using a normal form, you can establish the level of normalization required to perform on the database. There are three forms that you will use for normalizing databases: the first form, the second form, and the third form.

Every subsequent form that you use will rely on the techniques that you used on the form preceding it. You should ensure that you have used the right form

before you move on. For example, you cannot skip from the first form to the third form without doing the second form.

### **First Form**

The goal of using this form is to take the data and segregate it into tables. Once the tables are designed, the user will be able to assign a primary key to each table or group of tables. To attain this first form, you will divide up the data into small units, and each one needs to have a primary key and a lack of redundant data.

### **Second Form**

The goal of using the second form is to find the data that is at least partially reliant on those primary keys. Then, this data can be transferred over to a new table as well. This will help to sort out the important information and leave behind redundant information or other unnecessary things.

### **Third Form**

The goal of the third form is to eliminate the information that does not depend on any of your primary keys. This will help to dispose of the information that is in the way and slowing down the computer, and you will also discard the redundant and unneeded information along the way.

### **Naming Conventions**

When you are working on the normalization process, you need to be particular and organized with your naming conventions. Make use of unique names that will enable you to store and then later retrieve your data. You should choose names that are relevant to the information that you are working on in some way so that it is easier to remember these names in the future. This will help keep things organized and avoid confusion in the database.

### **Benefits of Normalizing Your Database**

We have spent some time talking about normalization in your database, but what, exactly, are the benefits? Why should you go through this whole process simply to clean out the database that your customers are using? Would it still work just fine to leave the information as is and let the search

sift through the redundancies and other information that you don't need? Here are some of the beneficial reasons why you should utilize normalization and avoid letting your database become inefficient:

- Keeps the database organized and easier to use
- Reducing repetitive and redundant information
- Improves security for the whole system and its users
- Improves flexibility in your database design
- Ensures consistency within the database

It may seem like a hassle to go through and normalize the database, but it truly makes the whole experience better. Your customers will have an easier time finding the information that they want, the searching and purchasing process will become more streamlined, and your security will be top of the line.

Despite the many benefits, there is one downside to normalization. This process does reduce the performance of the database in some cases. A normalized database will require more input/output, processing power, and memory to get the work done. Once normalized, your database is going to need to merge data and find the required tables to get anything done. While this can help make the database system more effective, it is still important to be aware of it.

## **Denormalization**

Another process that you should be aware of is denormalization. This allows you to take a normalized database and change it to ensure that the database has the capability of accepting repetition. The process is important in some instances to increase how well the database can perform. While there are some benefits to using the normalization process, it is bound to slow down the database system simply because it is working through so many automated functions. Depending on the situation, it could be better to have this redundant information rather than work with a system that is too slow.

Normalization of your database has many great benefits and it is pretty easy to set it all up. You just need to teach the database to get rid of information that it finds repetitive or that could be causing some of the issues within your

system. This can help to provide more consistency, flexibility, and security throughout the whole system.

## Database Normal Forms

We briefly touched on the topic of normal forms, and in this section, we will look at them in more detail. To normalize a database, you need to ensure that a certain normal form is achieved. A database is said to be in a particular normal form if it adheres to a specific set of rules. A database can have six normal forms, which are denoted as 1NF, 2NF, 3NF, 4NF, 5NF, and 6NF. The higher the normal form, the more a database is normalized. Most of the real-world databases are in third normal form (3NF). You have to start with 1NF and work your way up to the higher normal forms. In this chapter, we will delve into the different types of normal forms.

### First Normal Form (1NF)

A database in 1NF adheres to the following rules:

#### Atomic Column Values

All the columns in the table should contain atomic values. This means that there should be no column that contains more than one value. The following table, which contains multiple names in the PatientName column, does not have atomic column values

PatientName	DepName
Jane, Alan, Elizabeth	Cardiology
Mark, Royce	Pathology

A downside to the table shown above is that you cannot perform CRUD operations on this database. For instance, you cannot delete the record of Jane alone; you would have to delete all the records in the Cardiology department. Similarly, you cannot update the department name for Mark without also updating it for Royce.

## No Repeated Column Groups

Repeated column groups are a group of columns that have similar data. In the following table, the PatientName1, PatientName2, and PatientName3 columns are considered repeated columns since all of them serve to store names of patients.

PatientName1	PatientName2	PatientName3	DepName
Jane	Alan	Elizabeth	Cardiology
Mark	Royce		Pathology

This approach is also not ideal since, if we have to add more patient names, we will have to add more and more columns. Similarly, if one department has fewer patients than the other, the patient name columns for the former will have empty records, which leads to a huge waste of storage space.

## Unique Identifier for Each Record

Each record in the table must have a unique identifier. A unique identifier is also known as a primary key and the column that contains the primary key is called the primary key column. The primary key column must have unique values. For instance, in the following table, PatientID is the primary key column.

PatientID	PatientName	PatientAge	PatientGender
1	Jane	45	Female
2	Mark	54	Male
3	Alan	65	Male
4	Royce	21	Male
5	Elizabeth	27	Female

If a database adheres to all of these conditions, it is considered to be in first normal form. You need to check the finer details to be able to establish this aspect.

## Second Normal Form (2NF)

For a database to be in second normal form, it must adhere to the following three conditions:

- Adheres to All the Conditions That Denote First Normal Form
- No Redundant Data in Any Column Except the Foreign Key Column
- Tables Should Relate to Each Other via Foreign Keys

Take a look at the following table:

PatientID	Patient Name	Patient Age	Patient Gender	DepName	DepHead	NoOfBeds
1	Jane	45	Female	Cardiology	Dr. Simon	100
2	Mark	54	Male	Pathology	Dr. Greg	150
3	Alan	65	Male	Cardiology	Dr. Simon	100
4	Royce	21	Male	Pathology	Dr. Greg	150
5	Elizabeth	27	Female	Cardiology	Dr. Simon	100

The above table is in 1NF since the column values are atomic. There is no existence of repeated column sets, and there is a primary key that can identify each of the records. The primary key is PatientID.

Some columns, like the last three columns, contain redundant values. Therefore, it is not in 2NF yet. The redundant columns should be grouped together to form a new table. The above table can be divided into two new ones: The Patient Table and the Department Table. The Patient Table will contain the following columns: PatientID, PatientName, PatientAge, and PatientGender. The Department Table will have an ID column, and the DepName, DepHead, and NoOfBeds columns.

--	--	--	--

ID	DepName	DepHead	NoOfBeds
1	Cardiology	Dr. Simon	100
2	Pathology	Dr. Greg	150

Now that we are creating two tables, the third condition of 2NF requires that we create a relationship between the two tables using a foreign key. Here, we know that one department can have many patients. This means that we should add a foreign key column to the Patient table that refers to the ID column of the Department table. The Department table will not change from what is shown above, and the updated Patient table will look like this:

#### **Patient Table**

PatientID	PatientName	PatientAge	PatientGender	DepID
1	Jane	45	Female	1
2	Mark	54	Male	2
3	Alan	65	Male	1
4	Royce	21	Male	2
5	Elizabeth	27	Female	1

In the above table, the DepID column is the foreign key column, and it references the ID column of the Department Table.

#### **Third Normal Form (3NF)**

You need to be careful and know when a database is in 2NF. To be considered 3NF, it must adhere to the following conditions:

- Should Satisfy All the Rules of the Second Normal Form

- All Columns in the Tables Fully Depend on the Primary Key Column

Take a look at the following table:

PatientID	Patient Name	Patient Age	Patient Gender	DepName	DepHead	NoOfBeds
1	Jane	45	Female	Cardiology	Dr. Simon	100
2	Mark	54	Male	Pathology	Dr. Greg	150
3	Alan	65	Male	Cardiology	Dr. Simon	100
4	Royce	21	Male	Pathology	Dr. Greg	150
5	Elizabeth	27	Female	Cardiology	Dr. Simon	100

In the above table, the DepHead and NoOfBeds columns are fully dependent on the primary key column, PatientID. They are also dependent on the DepName column. If the value in the DepName column changes, the values in the DepHead and NoOfBeds columns also change. A solution to this problem is that all the columns that depend on some column other than the primary key column should be moved into a new table with the column on which they depend. After that, the relation between the two tables can be implemented via foreign keys as shown below:

### **Patient Table**

PatientID	PatientName	PatientAge	PatientGender	DepID
1	Jane	45	Female	1
2	Mark	54	Male	2
3	Alan	65	Male	1
4	Royce	21	Male	2



5	Elizabeth	27	Female	1
---	-----------	----	--------	---

### Department Table

ID	DepName	DepHead	NoOfBeds
1	Cardiology	Dr. Simon	100
2	Pathology	Dr. Greg	150

### Boyce-Codd Normal Form (BCNF)

This normal form, also known as 3.5 Normal Form, is an extension of 3NF. It is considered to be a stricter version of 3NF, in which records within a table are considered unique. These unique values are based upon a composite key, which is created by a combination of columns. Though, this does not always apply or need to be applied for every table because sometimes the data in a table does not need to be normalized up to BCNF.

### Fourth Normal Form (4NF)

For this step, the previous form, BCNF, must be satisfied. This particular form deals with isolating independent multi-valued dependencies, in which one specific value in a column has multiple values dependent upon it. You'd most likely see this particular value several times in a table.

### Fifth Normal Form (5NF)

This is the last step in normalization. The previous normal form, 4NF, must be satisfied before this can be applied. This particular form deals with multi-valued relationships being associated with one another and isolating said relationships

# CHAPTER 19:

## Real-World Uses

Throughout this book, you had the first-hand experience of using SQL in a stand-alone environment. For instance, you created databases, tables, populated tables, pivoted tables, cloned tables, used stored procedures, etc. You also performed various operations on the tables to manipulate the data the way the data is viewed to achieve various outcomes.

SQL syntax uses basic English language and the basics of SQL is quite easy to learn and understand. As there are different SQL applications it can be quite a difficult language to master. Although SQL is a language you can write code in one SQL application and use it in another, some applications may not support various functions. Some applications may also vary slightly in how they use a statement, thus, SQL can, at times, seem like a difficult language to master.

In SQL you can reuse scripts, which ensures you only ever have to create a complex script once and never have to rebuild it from scratch. You can store the script as a *stored procedure* and access it at any time you may need to use it. Triggers can also be set to automatically run the script for you.

### SQL in an Application

You will find that SQL has a few weaknesses along with all its strengths. One of these weaknesses is that it is not a procedural language. SQL must be combined with a procedural language like Pascal, C, FORTRAN, COBOL, Visual Basic, Java, or C++ to be used in an application. By combining SQL

with one of these procedural languages, can overcome some of the language limitations, especially when creating an application.

SQL working alongside a procedural language allows for the creation of powerful applications with a wide range of capabilities. RDBMS has long since replaced the physical storage cabinets with paper files. These paper files needed large filing facilities or archives in which to store them. Now relational databases are used anywhere information is stored or retrieved, without the need for a person to physically go retrieve a paper file from reams of files. With the use of an RDBMS database you can retrieve data anytime, from anywhere, and a lot faster.

With the rise of RDBMS databases, there are now large platforms such as Wikipedia, and other informational based sites that you get information from. There is not much you cannot find on the Internet today, and all these sites use some form of RDBMS.

Sites like Amazon, Facebook, and even Netflix also use a database back-end in which their user base, products, etc. are stored. Database systems run on the back-end of these sites and also handle transactions such as orders and payments.

Example of industry or organizations that use RDBMS are:

- Banks are a prime example of an organization that uses databases. They are used for payment transactions, to manage user funds, deposits, and so on.
- Retail industries are another prime candidate for using RDBMS to store product information, inventory, sales transactions, customer bases, etc.
- Medical practices utilize databases for patient information, prescription medication, appointments, and other information.
- Environmental organizations use databases to store information such as weather patterns, land corrosion, environmental pollution, and various other statistics.
- Governmental agencies use RDBMS to store population statistics, resource utilization, and various other statistics about areas, people, and so on.

- Many, many other organizations use RDBMS daily such as schools, military, and even NASA.

Instead of trying to list all the applications RDBMS covers in this day and age, there are not many places or organizations that are not using a form of RDBMS.

RDBMS is not only a way of maintaining, organizing, and using data at the organization level but it also allows for an organization's customer base to interact with it. Customers have access to their profile information, payment methods, order, deliveries, etc. They can also manage their own data. In the medical field, this also means a customer has access to their patient records, medication history, and quick access to their medical advisor.

A digital database can leverage mobile applications. This expands on the opportunities for RDBMS including new software platforms that use databases daily. The use of mobile applications paired with RDBMS has seen exponential growth in the capturing of data. In turn, this has led to more development opportunities as the need for RDBMS is fast expanding to keep up with technology. All you have to do is think along the lines of email apps, chat apps, and social media apps that are run on mobile devices daily.

New types of databases mean a call for new types of development, which in turn means job opportunities. It is no longer only IT specialists who are becoming database developers. With programs that are becoming easier to use because of clever APIs and using basic English to write scripts, it is getting easier for anyone to learn to code. Python is a language that has great API's, and it is also not that difficult to learn. It is also free to download and use. As such, someone with a finance background could easily pick up the language and write a program to suit their needs. Incorporating an SQL database with a bit of learning time is just as cost-effective to do.

In a real-world situation, multiple users access a database at the same time. Some databases do not handle such a high level of concurrency and the result can be corrupted or loss of data. SQL uses transactions to control atomicity, consistency, isolation, and durability to enable it to manage multiple uses without instability, data corruption, or data loss.

T-SQL statements are a sequence of SQL transactions that combine logically and complete an operation that would otherwise introduce database inconsistency.

Atomicity is a property that mimics a container that stores transaction statements. A successful statement means that all transactions have been completed. The entire operation will fail at any time during the statement a condition is not met and it will not process any further. If the process fails SQL will automatically roll-back to its previous state before the transaction was executed. Only once a row, or a page-wide lock is in place will a transaction be executed. While the transaction is locked it will prevent the object from being updated or manipulated by another user. The object gets reserved for the user utilizing it until their transaction has been completed.

Any attempts to use the object by another instance will fail with a “data locked” warning. However, even when an object is locked, another user can work on it. Their position to update will be placed in a queue until the object becomes free.

Using transactions to transform data enables a database to efficiently move from one consistent state to a new one. Always keep in mind and understand that transactions can modify more than one database at a time. For instance, if you change data in a primary key or foreign key field, make sure to simultaneously update the data with these key effects. If this is not done there will be inconsistencies in the SQL data. Transactions are vital to being able to update multiple tables consecutively.

The transformation of transactions is what reinforces isolation and is a property that prevents concurrent interactions from creating interference between them. This means that simultaneous transactions can take place at the same time, only one of them will be successful, while the other will fail or be queued.

Until transactions are complete they remain invisible and the first transaction completed is the one that will be processed first. Historically, failed transactions would either be deleted or the user would decide to try run the process again or end it. Most systems will now queue the information until it

can try to rerun it. Only after a specified number of failed attempts will the process fail outright. This process is what lends more durability and stability to transactions which are handy in the case of a sudden system outage or power failure. In a worst-case scenario, all that would need to be done is to rerun the script. The system will automatically roll-back to before the execution of the failed transaction.

Any transactions that have failed to complete are retained. Rollbacks are accomplished by the A database engine will use transactional logs to identify what the state of data was before running the failed transaction. It will use this reference to match to a previous state and roll it back to that point.

## **Database Locks**

There are many variations of a database lock which include different locking properties that can achieve varied database lock results.

Some common properties of a database lock may include:

### **Mode**

A lock mode is the type of database lock being applied. There are a few different types of lock modes including:

- Shared lock mode — While the record is locked this mode allows data reads.
- Exclusive lock mode — Used with DML to provide an exclusive page or row to allow for data modification. It will render all access to the data inaccessible regardless of user rights while the data is being modified.
- Update lock — An update lock is a single object that allows data reads while the record is locked. The update lock will also review and determine the necessity of an exclusive lock. This is handy to make sure the exclusive lock releases a record after it has been committed. The update lock shares the function of both the exclusive lock and shared lock.

### **Granularity**

The final property of a lock, the granularity, specifies to what degree a resource is unavailable. Rows are the smallest object available for locking,

leaving the rest of the database available for manipulations. Pages, indexes, tables, extents, or the entire database are candidates for locking. An extent is a physical allocation of data, and the database engine will employ this lock if a table or index grows and more disk space is needed. Problems can arise from locks, such as lock escalation or deadlock, and we highly encourage readers to pursue a deeper understanding of how these function.

## **Duration**

One of the simplistic database lock properties to define is the duration property. The duration property of a database lock simply specifies the duration of the object lock.

It is important to note that the locks listed above are page-level locks. There are more advanced locks but those are beyond the scope of this book.

## **PL/SQL**

Oracle developed an extension for procedural instruction using SQL syntax called PL/SQL. As discussed at the beginning of this chapter, SQL is a non-procedural language and needs to be paired with a procedural language to create full applications. The PL/SQL extension from Oracle expands the scope of SQL's capabilities as a programming language.

SQL, with the use of PL/SQL code, now offers more functions, stored procedures, and triggers. Triggers are what enable SQL to perform specific transactions when specific conditions are met. These conditions are usually defined by the database administrator and work in conjunction with various notification alerts, such as alarms. PL/SQL was developed by Oracle to meet their particular RDBMS product requirements. PL/SQL came about to overcome the non-procedural limitations of SQL.

# CONCLUSION

By now you should have quite an in-depth knowledge of SQL. Because the code is simple and uses basic English statements, the coding may only slightly differ across various SQL server programs.

One of SQL's greatest strengths is its reusable code that can be easily adapted to different environments. So do not be afraid to try your new-found SQL skills in an environment other than MySQL. There are a few SQL databases that you can download and try. These databases include the likes of:

- Oracle Database
- SQL Server Management Studio(SSMS)
- MongoDB
- Microsoft SQL Server
- PostgreSQL
- MariaDB
- SQLite

The more SQL servers you try out and work on, the more diverse your SQL skills will become. There are also various SQL certification paths for those that want to add SQL as a certified skill to their resume to boost their IT careers or their salaries.

The following are a few SQL Certifications from Microsoft to consider:

- Microsoft Technology Associate (MTA): Database Fundamentals SQL
- Microsoft Certified Solutions Associate (MCSA): Database Administration
- Microsoft Certified Solutions Associate (MCSA): Database



## Development

- Microsoft Certified Solutions Expert (MCSE): Data Management and Analytics

There are also certifications from other RDBMS programs such as:

- MongoDB
  - MongoDB Certified Developer
  - MongoDB Certified DBA
- MySQL
  - Database Administration
  - Database Developer

You cannot go wrong by improving your SQL skills or getting certified in a world where data has become a valuable commodity.

Thank you for choosing *SQL: The Ultimate Intermediate Guide to Learning SQL Programming Step by Step* to help you further your SQL knowledge.

# REFERENCES

*Beginner's Guide to MySQL Storage Engines*. (2018, April 4). SiSense.  
<https://www.sisense.com/blog/beginners-guide-to-mysql-storage-engines/>

Edwards, B. (2016, October 13). *The Forgotten World of Dumb Terminals*. PC Magazine. <https://www.pcmag.com/news/the-forgotten-world-of-dumb-terminals>

Hosch, W. (n.d.). *Charles Bachman*. Britannica.  
<https://www.britannica.com/biography/Charles-William-Bachman>

Hosch, W. (n.d.). *Edgar Frank Codd*. Britannica.  
<https://www.britannica.com/biography/Edgar-Frank-Codd>

*Learn SQL*. (n.d.). Tutorialspoint. <https://www.tutorialspoint.com/sql/>

*Network Hardware*. (n.d.). BBC.  
<https://www.bbc.co.uk/bitesize/guides/zh4whyc/revision/7>

Relational Database. (n.d.). IBM.  
<https://www.ibm.com/ibm/history/ibm100/us/en/icons/reldb/>

Relational Database Definition. (n.d.). Omni-Sci.  
<https://www.omnisci.com/technical-glossary/relational-database>

*SQL - Constraints*. (n.d.). Tutorialspoint.  
<https://www.tutorialspoint.com/sql/sql-constraints.htm#:~:text=Constraints%20are%20the%20rules%20enforce>

*SQL Injections*. (n.d.) PortSwigger. <https://portswigger.net/web-security/sql-injection>

*SQL Tutorial*. (n.d.). w3schools. <https://www.w3schools.com/sql/default.asp>

# SQL

**The Ultimate Expert Guide To Learn Sql  
Programming Step By Step**

**© Copyright 2022 - All rights reserved.**

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

**Legal Notice:**

This book is copyright protected. It is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaged in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of the information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

# Table of Contents

[Introduction](#)

[Chapter 1:](#)

[Data Access with ODBC and JDBC](#)

[ODBC](#)

[ODBC Functional Components](#)

[Extensions](#)

[JDBC](#)

[Chapter 2:](#)

[Working with SQL and XML](#)

[The Relationship Between XML and SQL](#)

[XML Data Type](#)

[Mapping](#)

[Character Sets](#)

[Data Types](#)

[Identifiers](#)

[Tables](#)

[Null Values](#)

[The XML Schema](#)

[SQL Functions and XML Data](#)

[Converting XML Data Into SQL Tables](#)

[Chapter 3:](#)

[SQL and JSON](#)

[Combining JSON with SQL](#)

[The Data Model](#)

[Functions](#)

[Query Functions](#)

[Constructor Functions](#)

[Chapter 4:](#)

[Datasets and Cursors](#)

[Cursor Declaration](#)

[Updatability, Sensitivity, and Scrollability](#)

[Opening the Cursor](#)

[Fetching Data](#)

[Chapter 5:](#)  
[Procedural Capabilities](#)  
[Compound Statements](#)  
[Managing Conditions](#)  
[The Flow of Control Statements](#)  
[Stored Procedures and Functions](#)  
[Chapter 6:](#)  
[Collections](#)  
[Overview](#)  
[Associative Arrays](#)  
[Nested Tables](#)  
[Varrays](#)  
[Chapter 7:](#)  
[Advanced Interface Methods](#)  
[External Routines](#)  
[The Net Configurations](#)  
[External Programs](#)  
[Chapter 8:](#)  
[Large Objects](#)  
[LOB Data Types](#)  
[BLOB, CLOB, NCLOB, and BFILE](#)  
[Creating Large Object Types](#)  
[Managing Large Objects](#)  
[Chapter 9:](#)  
[Tuning and Compiling](#)  
[Compilation Methods](#)  
[Tuning PL/SQL Code](#)  
[Conclusion](#)  
[References](#)

# INTRODUCTION

This book assumes that you have a working knowledge of SQL and managing databases. The focus is to improve your skills and guide you through the most challenging aspects of working with SQL. To get the most out of the lessons, take an active approach to this book. There is only so much we can fit in these pages, so it pays to do further research on topics and methods that remain a little foggy. While basics and intermediate topics are easy to learn, the topics in this book will be hard. Don't fret, because when you find yourself struggling, that is often a sign that the learning process is taking place. You shouldn't find this discouraging. Anything worth knowing is always a bit difficult to understand, and that is what makes it valuable to know. That should motivate you and prepare your mind for the topic ahead. When you feel a little discomfort, you will know that it is normal; it will be a sign to step back, do some research, and digest.

You will be equipped with enough information for you to be able to carry out research. Take your time and allow yourself to digest the information in this book and do the work – it is the only way that the information contained here will stick with you and be of use to you.

I have done my job to make these topics as easy as possible for you to understand, but there is a lot of value in doing some work yourself(that's how concepts stick, after all).

We are going to tackle the following topics:

- Working with XML using SQL
- How XML relates to SQL
- How to convert XML data into SQL tables
- How to use SQL and JSON together to store data
- Cursors and how to use them
- Performance tuning

Let's not waste time and start with just a few rules: stick with it and keep practicing until you get it.



# CHAPTER 1:

## **Data Access with ODBC and JDBC**

Over the past decade, high internet speed and integration of computer systems has meant that we have become more connected than ever before. This increase in connectivity means various systems have to share data with one another, which makes database sharing systems an absolute must. More computer systems online necessitate that these networks be able to communicate through multiple networks sharing the same data, but this raises the need for compatibility between operating systems and hardware. SQL development circumvents these challenges by allowing the sharing of information, but it doesn't solve the problem completely—there are gaps.

The primary focus of SQL development has been compatibility. The problem is that SQL isn't standard as we would prefer. There are other ways of solving the problems SQL is trying to fix or plugging holes when they appear. Database management developers and other service providers use a variety of tools, services, and extensions in SQL in order to improve their technology or fill those gaps we talked about. This means that instead of the technology becoming more standardized, it becomes more customized and specific, raising the risk of it becoming incompatible with other systems. Businesses, governments, sole developers, and everyone in between have used the same methods and techniques to fix problems specific to their projects. This means they have come to rely on these extensions and services. Without them, their systems would collapse completely, and it would cost a whole lot of money to build them back up. With the risk of incompatibility high and with many developers reliant on extensions, there needs to be a

method that increases compatibility without adversely affecting performance. That tool is Open DataBase Connectivity (ODBC). It will be the focus of this chapter.

## **ODBC**

ODBC is an interface that allows for communication between an application and a database. This interface is standardized, meaning regardless of what technologies are employed by developers in the management of the database, apps in the front end will still be able to access it even when the app and the database itself are not strictly compatible. This is because ODBC works as a type of liaison. As long as the front end and the back end understand ODBC and follow its standards, they can work together. That is an ingenious solution: backend SQL developers can enhance their technologies with a wide variety of tools without worrying much about compatibility, and front-end developers can do the same.

In this framework, applications use an ODBC driver to gain access to the database. The driver's front end is connected to the application, ensuring that the ODBC standards are followed. This driver is the same and speaks the same language, regardless of the database engine that is fueling the back end. This helps back-end developers who usually develop to suit the database engine they rely on. This means back-end developers and front-end developers can be ignorant of the various technologies that furnish their counterparts because all they need to communicate with is the ODBC. They don't need to think about what's going on on the other side because it does not have any impact on their development. That is the magic of ODBC. Imagine the freedom that comes with this, and the amount of work it saves teams.

The ODBC interface can be described as a set of strict definitions that are to be followed. These definitions contain and spell out all the rules that both parties(back-end and front-end) need to play by for communication between the application and the database to be possible. Broken down further, the interface is a collection of a few elements: function call library, standard SQL syntax, SQL data types, error codes, and the standard protocol for

communicating with the database engine.

Here's how it works: to connect to the database engine, there is a function call that will run SQL statements and then communicate the output to the application. To do any operations, we need to write SQL statements in the form of an ODBC function call. What this means is that we have to use ODBC's SQL syntax. Amazingly, it does not matter what database engine we are working with, the process is the same.

## **ODBC Functional Components**

ODBC is built into four functional components known as layers. Each layer plays its role in facilitating communications between the front-end and the back-end. Now let's look at each of these layers briefly:

1. Application layer: this is the front-end. It is the part that a user communicates with. The application itself is not necessarily a part of ODBC; it is nonetheless part of the process because it decides to communicate with the interface to access the data or not. It does this through a component known as the driver manager which follows the ODBC standards.
2. The Driver Manager layer: A driver manager is a DLL, or dynamic link library. Its job is to retrieve the right driver systems data and manage the direction of function calls sent through the driver to the data source. In a nutshell, the driver manager processes ODBC calls. It can also detect and resolve errors when they occur.
3. Driver DLL layer: Because data sources differ a lot, we need a method of translating ODBC function calls to the language of the data source. It is the job of a driver DLL layer to translate these commands: it functions by accepting calls from the interface and translating them into code that can be read by the data source. When a request is fulfilled, it is also the job of the driver to translate results into standard ODBC. This is where the critical task of ODBC takes place. It is the core function. It is the one that allows any application and data source to communicate with each other.
4. Data Source Layer: This is the layer that is often as varied as the

databases. It can be a relational database system, an associated database that is on the same system as the app, a database hosted remotely, an indexed sequential access method file without a management system, and many more. Because of how varied data sources can be, each one requires its own custom driver.

In client/server systems, we will have application programming as the interface between the two components (ODBC and server system). For this reason, an ODBC driver will include an API that is either proprietary or standard. Proprietary APIs are built with a specific server back end in mind.

As mentioned earlier, the core of the interface is the driver, also known as the native driver when working with proprietary systems. The driver is the code that does the hard work. Responsiveness is becoming more and more important; the slower the system is, the worse the experience is for users, and the less likely they are to use the service again. What is neat about the driver is that it is optimized for minimum delay in data transfer. Performance is optimized on both sides: front end and back end. Information travels faster in both directions, which is what clever systems will be optimized to do.

If the client-server system will only need to access one data source type and there is no need to predict future use of other data types, you are best served using the native driver that comes with the database management system. But if it is clear that you will need to access future data that will be stored in a myriad of ways, you will be best served with the ODBC API from the start. This is why when you design your systems, you have to think ahead to avoid headaches – reworking an entire system brings a lot of pain.

You should know that ODBC drivers are optimized with specific data sources in mind. These optimizations depend on an identical interface plugged into the driver manager. This means if you have a non-optimized driver that wasn't made for a specific front end, it won't be as efficient as a native driver made for that specific front end. As you may know, when the first ODBC drivers were released, they didn't perform as well as they do now. This is because the latest versions are comparable to native drivers in how rigorous they are. We no longer have to trade in performance for standardization (stability and fluidity).

Although there are differences between database operations over the internet and those over the client-server system, users won't be able to notice the difference. The only time they can spot a difference is on the client-side itself or the app component. On an online connection, the client-side of the system is on the local machine, but communication with the server data is through HTTP standard protocol. This means if you are a client-side user, you will be able to access data that is stored online. You can, for instance, develop an app on your computer and have access to it through a mobile device.

## **Extensions**

Communication between a client system and a web server occurs through the HTTP protocol. These commands are converted into ODBC-viable SQL by a component called the server extension. When this happens, the server database can follow the SQL commands and the data source. The output, or result, is communicated back through the server and the server extension, which is then converted into a format that the web server can understand. This is how results are communicated online to the user making use of the front end client machine.

Having said this, various applications function on locally stored data, on a local or wide area network server, like Microsoft Access. It is also possible to access applications stored in a cloud through the web browser. Web browsers are so advanced nowadays that they can deliver great quality apps/user-friendly interfaces. But when you log into your browser, you can see that they are not optimized to act at the front end of a database. For any online app to interact with databases, there needs to be a functionality that allows this. That functionality is provided by client extensions which include Java applets, scripts, and ActiveX controls.

These extensions communicate with HTML through the HTTP protocol. Remember that HTML code which handles access to a database will be translated to SQL through a server extension before reaching the data source. Scripts are one of the best ways of developing a client extension. Languages like JavaScript allow us to have full control over anything that happens on the client-side. Through JavaScript, we can do validation checks on input fields, so that invalid entries can be rejected or edited before they are transmitted

online. Javascript saves a lot of time and reduces traffic because it can perform these tasks on the client-side. As a result, everybody's user experience is improved because there is not a lot of unnecessary traffic. A validation check can be performed on the server as well.

Intranet networks, which are often used by organizations for security and simplicity reasons, can also benefit from an ODBC. ODBC works the same way on a local network as it will on the internet. If you are using multiple data sources, users using browsers with the right extensions will be able to communicate through SQL, HTML, and ODBC phases respectively. The SQL will be converted into a native language of the database and executed.

## **JDBC**

Java DataBase Connectivity, JDBC, is an alternative to ODBC; they do the same thing but they are different in a few crucial ways. Like ODBC, the JDBC remains the same no matter the app and the database involved, but it requires the client-side applications to be written in Java only. So, you cannot have an application written in JavaScript, Python, C++, C#, or any other famous programming language. In short, ODBC accommodates a variety of programming languages and JDBC does not. JDBC and Java are also developed to work on networks, the intranet, or the internet. As we have seen, ODBC is capable of working on your local machine, offline.

As soon as a client is online, Java applets download to the client where it executes code. The applet is embedded within the HTML, and it works to give the client database functionality that allows it to access server data. The applet is a small program that lives on the server, and when a user logs in the program loads and runs on their machine. Applets are made to execute functions in a closed area, on the computer's memory, and they cannot affect anything that lies beyond their area. These restrictions are embedded in the design of applets so that dangerous applets can not damage the system, corrupt information, or access private information.

Using applets is not only safe, they are updated often. Users don't have to worry about downloading or updating to the latest version because the applet is loaded each time the client makes use of the applet. This means you don't

have to worry about compatibility loss, because users will always be up to date. This minimizes your concerns as a developer because the only thing you should concern yourself with is whether the applet is compatible with your server configuration. Browsers and users will always be taken care of because every time they use your service, they will have the compatible latest version of the applet loaded on their machine.

Java is a powerful language that allows developers to write complex applications that can work with any client-server database. JDBC works with applications written in Java to give them access to databases. JDBC is the ODBC of Java applications, but Java applications will not perform the way C++ applications do. When a system is on the internet or intranet there are a lot of operating conditions one has to deal with when compared to a client-server system, but this shouldn't discourage developers from using Java. An application client-side is the browser that does not have much computing power. This can be optimized to allow powerful database processes to be performed. Java applets provide that much-needed processing power.

It should be said that whenever a user downloads something from a server there is risk involved, especially when they download from an unknown server. But Java applet's selling point is in its inherent security, so the risk is minimized. So as a rule one should never download data from an unknown server.

You should know that JDBC passes SQL statements the same way as ODBC, both ways. This means you can have all the things you have when working with ODBC: result sets, error messages, etc. JDBC also allows an applet writer to write to the standard interfaces without learning anything about the database. JDBC can also work on client-server environments, despite being made for the internet; the application will be able to access the database through the JDBC interface.

# CHAPTER 2:

## **Working with SQL and XML**

Since 2008, eXtensible Markup Language has been supported by SQL. XML files have become the standard way of transferring data between various platforms. The language lets you store data and share it with any person, no matter what operating system, hardware, or application environment they are using. It connects different systems, and that's why it is important to learn about SQL and XML.

### **The Relationship Between XML and SQL**

Like HTML, XML is a markup language. This just means it is not a programming language like Python or Java, which makes use of things like functions, methods, and can express logic and perform tasks. Essentially, all programming languages are command executors. They are not really aware of the data they manipulate. XML, on the other hand, is “aware” of the data it stores. HTML deals with text formatting and documented-related graphics; XML manages the content’s structure. However, XML does not handle how the data is formatted. For that you would need a stylesheet. Stylesheets control the formatting of both XML and HTML. XML provides the document’s structure through what is called the XML schema. Schema is metadata that describes the concepts of the document, the position of various elements, and their order, and what data type they are. So, it is not just a markup language like HTML is.

There are two methods one can use to structure, save, and retrieve data for use. The first one is provided by SQL. It offers a variety of options for



dealing with numeric and textual data. Furthermore, data can be divided into categories of data type and size. As you might remember, one of the purposes of SQL is to offer a standard method of operating with and maintaining data that is stored in relational databases. The second method is provided by XML. XML is better equipped at dealing with free form data, which cannot be easily categorized. One of the reasons we have XML is because developers wanted a standard way of performing data transfers between different computer systems.

As you can see, SQL and XML can go together as they can supplement each other. Using them together provides you with greater flexibility and distribution. If freedom is what you value, this combination is your best bet.

## **XML Data Type**

Correct standard XML data type implementations can deposit and operate XML data without converting it into SQL data type. The XML data types behave like user-defined types with all their subtypes. SQL and XML can work together in remarkable ways. For instance, applications can perform XML operations on SQL data and the other way around. You can have an XML type column with columns of predefined types, then have joint operation within the query's WHERE clause. Relational databases and database management systems will find the best way of running the query and executing it.

It can be tricky to know when it is best to store data in XML. There are few things you have to consider before storing data in XML. The following considerations are the best guide.

1. If you are storing information to access it in the future. Of course all information is stored with the hope of using it later. What's meant here is if the information is integral to the functioning of a process or function down the line.
2. Whenever the project you are working on requires strong data typing within your SQL statement. XML has an added advantage of making data values valid XML values instead of just strings.
3. If you know you will need to query an entire XML document. You should keep in mind that some implementations will include the

expansion of the EXTRACT operator so that you are allowed to extract specific information from the XML file.

4. If you want your tech to be future proof. Many of the ongoing optimizations offer XML data type support, and compatibility is still one of the most important things in any design. So, when you store data in XML, you know compatibility is guaranteed.

Below is an example of an XML data type:

```
CREATE TABLE ORDER (  
  CustomerName CHAR (40) NOT NULL,  
  Address CHAR (50),  
  City CHAR (30),  
  State CHAR (2),  
  ZIPCode CHAR (10),  
  Phone CHAR (13),  
  Email CHAR (40),  
  Comments XML(SEQUENCE) );
```

This is an example of an SQL statement that allows for the storage of an XML file inside the Comments column (a part of the ORDER table).

While SQL allows you to work with XML data types, it does not mean you should use this option all the time. There are many cases where it is best to avoid it. Relational databases are well suited to remain as they are in most cases. Below are two examples where that makes sense:

- Whenever data needs to be presented in rational architecture like tables.
- We want to update parts of a file and not the entire file.

## Mapping

To transfer data from an XML file to an SQL database, database components need to be translated to their XML equivalent. This happens in two ways. Let us look at this.

### Character Sets

Support of character sets will depend on the SQL implementation you are working with. For instance, DB2 implementations offer character sets support but SQL does not. An SQL server might provide support for certain character

sets that are not supported on Oracle. Some character sets will be supported by all implementations. This is why you have to think carefully about the character sets you use, especially if you plan to move your database to another management system. Support issues can cause a lot of problems.

XML only supports one character set: Unicode. On the face of it, it appears very limiting and you might get discouraged. But when it comes to SQL and XML, this is an advantage because it allows for data transfer between the SQL implementation and XML. All that the relational database has to do is to provide a mapping definition between strings of all character sets and Unicode, and vice versa.

## Data Types

An SQL data type has to be mapped to its nearest XML schema data type, according to the SQL standard. This way the values accepted by the SQL data type will be accepted by the XML data type. This means values that are rejected by SQL data type will be rejected by the XML schema, too. However, there are certain XML elements, like `minInclusive` and `maxInclusive`, that may limit the values the XML schema will accept. For example, if we have an SQL type that limits the `INTEGER` type's values to this range “-2246392734<value<2246392733”, then we need to define the `maxInclusive` value in XML to 2246392733 and the `minInclusive` value to -2246392734. Let's see how this mapping works:

```
<xsd:simpleType>
  <xsd:restriction base="xsd:integer"/>
  <xsd:maxInclusive value="2246392733"/>
  <xsd:minInclusive value="-2246392734"/>
  <xsd:annotation>
    <sqlxml:sqltype name="INTEGER"/>
  </xsd:annotation>
</xsd:restriction>
</xsd:simpleType>
```

(Adapted from source:

[https://www.w3.org/2001/sw/rdb2rdf/wiki/Mapping\\_SQL\\_datatypes\\_to\\_XMI](https://www.w3.org/2001/sw/rdb2rdf/wiki/Mapping_SQL_datatypes_to_XMI)  
retrieved in December 2019)

Take note of the annotations where data from the SQL type is retained so it

can be used by XML.

## Identifiers

XML is far more strict when it comes to characters that are allowed within identifiers. So characters that are not accepted in SQL but rejected in XML will have to be mapped so that they will be accepted. SQL can accept special characters like (\$, &, #) that are usually enclosed in double quotations, but these characters are not accepted in XML. In XML names, characters can't be used because those characters are already reserved. This means any SQL identifiers which start with these characters will have to be edited in a way that allows them to be accepted.

Thankfully, there is an established standard that fixes this problem. SQL identifiers will be translated to Unicode, and all the identifiers which are accepted by XML Names will remain the same. Unacceptable SQL character that can't make an XML Name are translated to hexadecimal code: like "xNNNN", where N stands for uppercase digits; an underscore is translated to "\_x006F\_;" a colon to "\_x00#". Any SQL identifier that starts with x, m, or i will be prefixed with "\_XFFFF\_".

The translation of characters from XML to SQL is simple. XML characters have to be scanned and when a sequence is found, it will be replaced with matches. So XML "\_FFFF\_" will be ignored.

It is simple to follow and easy to have SQL identifiers mapped to an XML Name, and vice versa.

## Tables

XML has table mapping. Tables are mapped in schema and it preserves privileges. Anyone can SELECT privilege enabled for columns and they will be able to map those columns to the XML file. The mapping process will create two files: one will have the information in the table and the other will have the XML schema which carries metadata/describes the file. Let's take a look at an example of how to map an SQL table to an XML file:

```
<CLIENT>
<row>
<FirstName>John</FirstName>
```

```
<LastName>Smith</LastName>
<City>Seattle</City>
<AreaCode>345</AreaCode>
<Telephone>555-5511</Telephone>
</row>
<row>
<FirstName>Frodo</FirstName>
<LastName>Baggins</LastName>
<City>Bree</City>
<AreaCode>345</AreaCode>
<Telephone>555-3344</Telephone>
</row>
</CLIENT>
```

(Adapted from source:

[https://www.w3.org/2001/sw/rdb2rdf/wiki/Mapping\\_SQL\\_datatypes\\_to\\_XML](https://www.w3.org/2001/sw/rdb2rdf/wiki/Mapping_SQL_datatypes_to_XML)  
retrieved in December 2019)

In the example above, the file base element receives the name of the table. Each row includes the “row” tag which holds the column elements. Each element has the same name as the matching column from the initial table and they contain a relevant value.

## Null Values

You will find SQL data often contains null values. You will need to decide how you want them stored in an XML. There are two options you can try. The first one is this:

```
<row>
<FirstName>John</FirstName>
<LastName>Smith</LastName>
<AreaCode>345</AreaCode>
<Telephone>555-3344</Telephone>
</row>
```

It doesn't show you much. That's because the row with a null value doesn't have a reference. In the second option you have to use `xsi:nil="true"` as an attribute for elements that contain null values. That would look something like this:

```
<row>
<FirstName>John</FirstName>
<LastName>Smith</LastName>
```

```
<City xsi:nil="true"/>
<AreaCode>345</AreaCode>
<Telephone>555-3344</Telephone>
</row>
```

(Adapted from source:

[https://www.w3.org/2001/sw/rdb2rdf/wiki/Mapping\\_SQL\\_datatypes\\_to\\_XMI](https://www.w3.org/2001/sw/rdb2rdf/wiki/Mapping_SQL_datatypes_to_XMI)  
retrieved in December 2019)

## The XML Schema

Once we have mapped SQL to XML, we will end up with a generated file which holds all the information. Like I have said, there will be a second document which contains schema info or data about the data. I wanted to show you how this looks with the real example below. Below is a CLIENT, the one we have been using as an example all this time:

```
<xsd:schema>
<xsd:simpleType name="CHAR_15">
<xsd:restriction base="xsd:string">
<xsd:length value = "15"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="CHAR_25">
<xsd:restriction base="xsd:string">
<xsd:length value = "25"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="CHAR_3">
<xsd:restriction base="xsd:string">
<xsd:length value = "3"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="CHAR_8">
<xsd:restriction base="xsd:string">
<xsd:length value = "8"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:sequence>
<xsd:element name="FirstName" type="CHAR_15"/>
<xsd:element name="LastName" type="CHAR_25"/>
<xsd:element name="City" type="CHAR_25 nillable="true"/>
<xsd:element name="AreaCode" type="CHAR_3" nillable="true"/>
<xsd:element name="Telephone" type="CHAR_8" nillable="true"/>
</xsd:sequence>
```

```
</xsd:schema>
```

(Source: <http://anyflip.com/dmjz/jouf/basic/551-600> retrieved in December 2019)

As you can see, the schema uses the “nil” attribute to deal with null values. Another option dealing with them looks something like this:

```
<xsd:element name="City" type="CHAR_25" minOccurs="0"/>
```

## SQL Functions and XML Data

There are SQL operators and functions which are capable of producing XML results when we run them on the SQL database. There are also operators and functions that can be applied to XML data. We are going to spend time looking at these so that you can see how they work. To get more information about them or to find solutions to problems that deal with your specific problem, I suggest you go see the documentation; it's not hard to digest. Some of the functions below will depend on XQuery, a query language that works exclusively with XML data.

1. XMLDOCUMENT is an XML operator that accepts a value input and returns another value as an output. The output will be an XML value in the form of a document node. It is generated by following the rules of the Xquery document constructor. If you're curious about XQuery, you can find resources online that will help you get acquainted. I won't discuss it any further here as it is beyond the scope of this book.
2. XMLELEMENT converts any relational value into its XML counterpart. When you introduce an operator within a SELECT statement for extracting data from an SQL database and translate the data into XML so that you can publish it, this operator will help you.
3. XMLFOREST is an XML operator used to create a list, known as forest, that is a collection of XML elements extracted from relational values. Every value will create its own element.
4. XMLAGG is an aggregate function. It accepts XML files or sections of them in the form of an input then it generates another XML file as the output. This function will contain a forest of

components.

5. XMLCONCAT is an alternative to creating lists of elements. You can use it to make lists, forest, through the concatenation of XML arguments, as the name suggests.
6. XMLQUERY is used to evaluate an XQuery argument and then return a result that is specified by the SQL program.
7. XMLCOMMENT, as the name suggests, lets you create XML comments.
8. XMLCAST is similar to the SQL CAST functions, but it has a few limitations. This function will allow your program to cast any value from an XML data type to another. It also works well with the SQL data type. It can also be cast from SQL to XML.

The functions above are the most frequently used, or the ones you will find useful in plenty of scenarios. You can learn about them, as well as others from the XML documentation; I promise you that the documentation is well structured and easy to understand.

## Converting XML Data Into SQL Tables

It used to be that we would spend time converting SQL data into XML so that we could use it online. It was a simpler option, and using it has opened us up to other possibilities. Today, we can translate XML data into SQL tables and query them with old SQL statements. This might be a desirable option depending on your project. The process of converting XML data into SQL tables is handled by a pseudo-function called the XMLTABLE. Below is the function's syntax:

```
XMLTABLE ( [namespace-declaration,]  
XQuery-expression  
[PASSING argument-list]  
COLUMNS XMLtbl-column-definitions  
value-expression AS identifier  
column-name FOR ORDINALITY  
column-name data-type  
[BY REF | BY VALUE]  
[default-clause]  
[PATH XQuery-expression]
```

(Source:



[https://www.ibm.com/support/knowledgecenter/en/SSEPGG\\_11.1.0/com.ibm](https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm)  
retrieved in December 2019)

Seeing the syntax is all well and good, but we could do better with a real-world example. But before we do that, there is something you should know. The pseudo function puts the XML data into an SQL pseudo-table. What's a pseudo-table? I know it's strange, but a pseudo-table is basically a table that exists momentarily, so it is not really real. It is possible to use the CREATE TABLE statement on it and add it to the XML file though. Here is our example of the pseudo-function:

```
SELECT client phone.*
FROM
clients_xml , XMLTABLE(
'for $m in $col/client return $m'
PASSING
clients_xml.client AS "col"
COLUMNS
"ClientName" CHARACTER (30) PATH 'ClientName' ,
"Phone" CHARACTER (13) PATH 'phone')
AS clientphone
```

The SQL standard allows us to store information in a structured manner. Users can easily maintain the data and access it as needed. XML is advantageous because it allows for communications between systems that would generally be incompatible with each other, expanding reach, audience, and utility. XML also makes it easy for us to work with. We can easily translate it to SQL data as needed. The methods, functions, and techniques above have illustrated this to us. The advantages of using SQL with XML are plenty, and they are also pain-free. They make it easy to move data and maintain it without worrying about the compatibility of your project with other systems.

# CHAPTER 3:

## SQL and JSON

In the early stages of computer processing and basic programming, there wasn't a luxury of storing and maintaining data inside a database. Those early programmers came with an ingenious solution: they used flat files to store information. But these flat files had one drawback: they weren't able to hold data in a structured format. Databases were made to fix these problems, especially those with a hierarchical structure. But sooner these were replaced by relational databases which have now become the norm. Relational databases are still constructed with the same architecture from the 80s. To manipulate them we have been using SQL.

Recently, with the explosion of big data, varieties of non-relational database architectures have come to the scene. These new databases are now known as NoSQL database structures. They all have a special, specific new way of storing and handling data. This is because developers are confident they can store data in one type of NoSQL database and still be able to transfer it and manipulate it to another type of NoSQL database. They circumvent the issues of working with various file formats in different databases by establishing a universal file that all databases can understand and work with. This universal interchange format is known as JavaScript Object Notation, or JSON. As the name suggests, JSON is based on JavaScript, which is a programming language that has taken over the internet. As the "Object" in JSON suggests, we can translate information into objects; JSON objects, which look a lot like JS objects. JSON objects can be converted into a variety of things, like strings, numbers, arrays, null values, and more.

## **Combining JSON with SQL**

NoSQL databases are becoming very popular. This means it is becoming a necessity to transfer data from relational databases to NoSQL databases. Vast quantities of information stored over decades are indispensable. To use it, an application based on an SQL database has to find a way of using vast quantities of data stored in relational databases. We also have to think of applications that use relational databases and need information from a NoSQL database for it to be fully optimized. The good news is that SQL allows us to convert JSON data into a data type that SQL systems can understand. The same happens the other way around. Solutions like these have become more and more important the more interconnected we become. But to perform data conversion, we need a tool that can define all data objects and functions that are dependent on them.

To introduce JSON information inside a relational database, it has to be given a character or binary string. Both types are stored in a column and you can extract and operate on them by using SQL and JSON functions. For instance, when answering an SQL query, we can use functions to create JSON items no matter what the original data source is (JSON or SQL data). We can perform these actions because a specific path language is now part of SQL operators. We can query JSON.

### **The Data Model**

JSON and SQL store data using different processes, but sometimes information needs to be shared between both systems. To allow this we need to build a communications bridge between them. The SQL/JSON data model allows us to do this.

All JSON data can be found in arrays, objects, null values, true values, strings, numbers, and more. The obvious problem is that SQL does not have the equivalent of these aspects of JSON, and null values, strings, and numbers are not identical in both environments. JSON also does not have the equivalent of datetime found in SQL. These problems mean we need to define specific SQL/JSON objects which will be stored within the SQL database. Even when the data is stored in SQL, it will still be able to

communicate with the JSON information, no matter where it is stored.

For instance, JSON data can be parsed into SQL/JSON objects where they will be stored as binary or character strings. The object can be found in the following SQL/JSON forms: scalar, arrays, objects, and null. A scalar is any value that isn't null but belongs to any of these categories: boolean, datetime, numeric, or character string. The null object is any value that differs from any other value that belongs to SQL data types. This will make it different from other null values as well. An array is a list of SQL/JSON objects. Items in the array will be separated like below:

```
[ 4.213545, "meaning of life", true]
```

It's worth mentioning that arrays in SQL start with one (their index), but in the SQL/JSON model, just like in JavaScript, they start with zero.

The SQL/JSON object is a compilation of any item that is paired with a character string and an SQL/JSON item. In this instance, the string value is called the key and the value it points to is the bound value. They are often referred to as key/value pairs. The object below is an example of this.

```
{ "name" : "John Smith", "ID" : 042, "goal" : "instructions" }
```

The key will be the *string* before the colon, and after those we have the *value* that the key points to.

## Functions

All operations performed on the JSON data are performed using SQL/JSON functions. These functions are either query functions or constructor functions. The first one evaluates the path of language expressions on JSON values. Those values will generate SQL/JSON values that will be translated into SQL. Constructor functions generate JSON values using SQL types' values.

There is more than one query function, and they all follow the same syntax. Below is an example of what that might look like. Keep in mind you can summarize this into a path expression; a JSON value would be the input and various parameters can be passed through the expression.

```
<JSON API common syntax> ::=  
<JSON context item> <comma>
```

```
<JSON path specification>  
[ AS <JSON table path name> ]  
[ <JSON passing clause> ]  
<JSON context item> ::= <JSON value expression>  
<JSON path specification> ::=  
<character string literal>  
<JSON passing clause> ::= PASSING <JSON argument>  
[ { <comma> <JSON argument> } ]  
<JSON argument> ::=  
<JSON value expression> AS <identifier>
```

(Source: <https://JSONapi.org/format/> retrieved in December 2019)

The expression type above is a string type. A JSON context object can be defined as a value expression with an extra input class that is present in the JSON's presentation format. The format describes the encoding so we can determine whether it's implementation-defined or Unicode format.

## Query Functions

There are four SQL/JSON query functions: JSON\_QUERY, JSON\_VALUE, JSON\_EXISTS and JSON\_TABLE. They compare path value expressions to the JSON values and then return SQL/JSON type values that are then translated to SQL.

Let's take a look at the JSON\_EXISTS function. Its job is to verify the JSON value to see if the path search condition is met. Here is the syntax:

```
<JSON exists predicate> ::=  
JSON_EXISTS <left paren>  
<JSON API common syntax>  
[ <JSON exists error behavior> ON ERROR ]  
<right paren>  
<JSON exists error behavior> ::=  
TRUE | FALSE | UNKNOWN | ERROR
```

(Source: [https://docs.oracle.com/en/database/oracle/database/18/adjsn/condition-JSON\\_EXISTS.html#GUID-8A0043D5-95F8-4918-9126-F86FB0E203F0](https://docs.oracle.com/en/database/oracle/database/18/adjsn/condition-JSON_EXISTS.html#GUID-8A0043D5-95F8-4918-9126-F86FB0E203F0) retrieved in December 2019)

The "ON ERROR" clause is optional. If you don't use it the system assumes that we have opted for "FALSE ON ERROR". It will analyze the path expression and only return a TRUE if at least one object is located.

The value function gets the SQL scalar from the JSON value. Below is the

syntax:

```
<JSON value function> ::=  
JSON VALUE <left paren>    <JSON API common syntax>  
[ <JSON returning clause> ]  
[ <JSON value empty behavior> ON EMPTY ]  
[ <JSON value error behavior ON ERROR ]  
<right paren>  
<JSON returning clause> ::=  
RETURNING <data type>  
<JSON value empty behavior> ::=  
ERROR | NULL | DEFAULT <value expression>  
<JSON value error behavior> ::=  
ERROR | NULL | DEFAULT <value expression>
```

(Source: [https://docs.oracle.com/en/database/oracle/oracle-database/18/adjsn/function-JSON\\_VALUE.html#GUID-0565F0EE-5F13-44DD-8321-2AC142959215](https://docs.oracle.com/en/database/oracle/oracle-database/18/adjsn/function-JSON_VALUE.html#GUID-0565F0EE-5F13-44DD-8321-2AC142959215) retrieved in December 2019)

In the “JSON\_EXISTS” example, rows are returned based on the “where” keyword inside the J column. The “JSON\_VALUE” function only returns a value when it is related to a target keyword. The JSON\_VALUE function returns characters that are implemented-defined by default; however, we can specify which data type we want with the “RETURNING” clause.

This function is best used to return a scalar from a SQL/JSON model, but it isn’t the best option when you are trying to retrieve arrays or other value objects. To return arrays and objects, the “JSON\_QUERY” function works best. Below is the JSON\_QUERY’s syntax.

```
<JSON query> ::=  
JSON_QUERY <left paren>  
<JSON API common syntax>  
[ <JSON output clause> ]  
[ <JSON query wrapper behavior> ]  
[ <JSON query quotes behavior> QUOTES  
[ ON SCALAR STRING ] ]  
[ <JSON query empty behavior> ON EMPTY ]  
[ <JSON query error behavior> ON ERROR ]  
<right paren>
```

(Source: <https://jsonapi.org/format/> retrieved in December 2019)

When JSON\_QUERY has on error and on empty clauses, they will be processed the same way as value functions but with one difference: the user

is allowed to specify how the error and empty clauses should behave. Working with a “WHERE JSON\_ EXISTS” clause will remove all rows without the key/value, and if there is an array wrapper involved, all array elements will be returned and put within square brackets.

The “JSON\_TABLE” function is perhaps the most sophisticated of all the query functions we have discussed so far. This function accepts JSON data as an input and then generates a relational output table based on it. Because of how complex this function is, it won't serve us well to go over the syntax. It can be overwhelming especially when it is your first time looking at it. It is best to have an entire chapter dedicated to it that talks about it briefly. Since I won't be doing this here, I think it is best to go to the official documentation and take your time to learn it. You should do this if you believe that it may be useful to you, and remember that you'll learn best when you practice.

## Constructor Functions

As the name suggests, constructor functions construct, so it won't come as a surprise that SQL/JSON constructor functions help set up JSON arrays and objects. They make use of data in relational tables to achieve this. When you compare them to query functions they act oppositely. Query functions fulfill requests and find things, but constructors build.

We will begin by taking a look at the JSON\_OBJECT function:

```
<JSON object constructor> ::=
JSON_OBJECT <left paren>
[ <JSON name and value> [ { <comma> <JSON name and value> } ... ]
[ <JSON constructor null clause> ]
[ <JSON key uniqueness constraint> ] ]
<right paren>
<JSON name and value> ::=
[KEY] <JSON name>
VALUE <JSON value expression> | <JSON name> <colon> <JSON value expression>
<JSON name> ::=
<character value expression>
<JSON constructor null clause> ::=
NULL ON NULL | ABSENT ON NULL <JSON key uniqueness constraint> ::=
WITH UNIQUE [ KEYS ] | WITHOUT UNIQUE [ KEYS ]
```

(Source:

[https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON\\_Cor](https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON_Cor)  
retrieved in December 2019)

When using these functions it is important to be aware of certain rules:

1. JSON value expressions can be null but not the name of the values.
2. An SQL/JSON null will occur if the JSON constructor clause is “NULL ON NULL”. If the clause is “ABSENT ON NULL” the key/value pair will not be included in the data model object created.
3. NULL ON NULL will be the default if we don’t have a null clause.

In some cases, it might be desirable to create a JSON item in the form of a relational table data aggregation. If we have two columns in the table, one containing values of the other names, we can use the OBJECTAGG function to create an object. Here's how the function will look:

```
<JSON object aggregate constructor> ::=  
JSON_OBJECTAGG <left paren>    <JSON name and value>  
[ <JSON constructor null clause> ]  
[ <JSON key uniqueness constraint> ]  
[ <JSON output clause> ]  
<right paren>
```

(Source:

[https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON\\_Cor](https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON_Cor)  
retrieved in December 2019)

Without a constructor class, the default will be “NULL ON NULL”.

If we want to construct a JSON array from a list of objects found in a relational database, we can use the ARRAY function. Here is how the ARRAY function looks:

```
<JSON array constructor> ::=  
<JSON array constructor by enumeration>  
| <JSON array constructor by query>  
<JSON array constructor by enumeration> ::=  
JSON_ARRAY <left paren>  
[ <JSON value expression> [ { <comma> <JSON value expression> }... ]  
<JSON constructor null clause> ] ]
```



```
<JSON output clause>  
<right paren> <JSON array constructor by query> ::=  
JSON_ARRAY <left paren>  
<query expression> [ <JSON input clause> ]  
[ <JSON constructor null clause> ]  
[ <JSON output clause> ]
```

(Source:

[https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON\\_Cor](https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON_Cor)  
retrieved in December 2019)

There are two types of the ARRAY function: one will generate the result by using an input list made of SQL values; the other will generate results by using the query expressions called inside the functions. If the constructor null clause is not included (which is optional), “ABSENT ON NULL” is the default. As you might have noticed, the object function works the opposite way.

If you want to create an item based on relational data aggregation, you might want to decide to use an array. ARRAY\_AGG is the function that helps you achieve this; this is part of SQL. Here’s how it looks:

```
<JSON array aggregate constructor> ::=  
JSON_ARRAYAGG <left paren> <JSON value expression>  
[ <JSON array aggregate order by clause> ]  
[ <JSON constructor null clause> ]  
[ <JSON output clause> ]  
<right paren>  
<JSON array aggregate order by clause> ::=  
ORDER BY <sort specification list>
```

(Source:

[https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON\\_Cor](https://docs.actian.com/actianx/11.1/index.html#page/SQLRef%2FJSON_Cor)  
retrieved in December 2019)

Just like before, if you don’t have a constructor null clause, “ABSENT ON NULL” will be the default. Using “array order by clause” will allow you to arrange the output items in a variety of ways you find desirable or appropriate (it works the same way that “ORDER BY” does in SQL).

Next, we have the “IS” JSON predicate used to verify whether a string type is valid JSON. Here’s how it looks:

```
<JSON predicate> ::=  
<string value expression> [ <JSON input clause> ]  
IS [NOT] JSON  
[ <JSON predicate type constraint> ]  
[ <JSON key uniqueness constraint> ]  
<JSON predicate type constraint> ::=  
VALUE | ARRAY | OBJECT | SCALAR
```

(Source:

[https://www.ibm.com/support/knowledgecenter/en/ssw\\_ibm\\_i\\_72/db2/rbafzis](https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/db2/rbafzis),  
retrieved in December 2019)

If an input clause is not specified, the default will be “FORMAT JSON”. There are optional key uniqueness constraints that without specification will set “WITHOUT UNIQUE KEYS” as default.

JSON nulls don’t work the same way SQL nulls do. For instance, take an SQL entity with a length of zero in the null value. This translates as undefined (lack of defined value). In JSON, however, null is seen as a real value--null is a string literal.

Nulls from SQL are different from those in JSON, which means a developer will have to think carefully about including them within a JSON item.

# CHAPTER 4:

## Datasets and Cursors

Most programming languages are incompatible with SQL. This poses a unique challenge for SQL developers. Take table data; SQL seeks to perform various operations on all the data at the same time, whereas most languages only process one row at a time. So developers have found a solution in cursors, which allows SQL to modify itself to work one row at a time. This allows SQL to work with other languages, therefore overcoming compatibility issues.

Cursors, as the name suggests, are pointers that point to a certain row in a table. Your activate operations will be performed on that specific row. Pointers are useful when you need data from a specific row or rows in a table. They also work when you need to vary the content in the row, and with a number of other operations like select, update, and delete. SQL can extract rows, but a more procedural language is appropriate at working with rows of content. Cursors allow SQL the ability to collaborate with a procedural language. This is illustrated when we put SQL inside a loop to go over one table row at a time. Here's how that would look:

```
EXEC SQL DECLARE CURSOR statement
EXEC SQL OPEN statement
Perform test for the end of the table
Procedural code
Initiate loop
Procedural code
EXEC SQL FETCH
Procedural code
Test for end of table
```

```
End loop  
EXEC SQL CLOSE statement  
Procedural code
```

(Source:

[https://docs.actian.com/ingres/10s/index.html#page/OpenSQL/Retrieve\\_the\\_F](https://docs.actian.com/ingres/10s/index.html#page/OpenSQL/Retrieve_the_F)  
retrieved in December 2019)

As you may have noticed, all the main SQL statements are there, including statements like to declare, open, fetch, and close. For the rest of the chapter we will go into more detail about workflow and these statements.

# Cursor Declaration

Just like you would with most languages, cursors need to be declared in the database management system to be used. In the previous example, the “declare cursor” statement is the one that is responsible for this. Note that the declaration itself does not perform a task, it announces the presence of the cursor. Declarations can also specify the query type. Let’s look at the syntax:

```
DECLARE cursor-name [<cursor sensitivity>]
[<cursor scrollability>]
CURSOR [<cursor holdability>]
[<cursor returnability>] FOR query expression
[ORDER BY order-by expression]
[FOR updatability expression] ;
```

(Source: <https://www.dummies.com/programming/sql/how-to-declare-a-sql-cursor/> retrieved in December 2019)

Cursors have to be identified by their name, and that name needs to be unique, especially when you have multiple cursors in a module. The name has to be indicative of its functions. Don't name your cursors something obscure like “AA”. The name has to be relevant and even descriptive so it makes it easier for you to work with and to remember what it points to.

When declaring a cursor keep in mind the following guidelines:

1. You need to set cursor sensitivity. You can set it to sensitive “asensitive”, or set it to select sensitive or insensitive. This is important because in some cases your statements might change the data that qualifies in some way. More on this later.
2. You should also think of cursor scrollability. You need to choose between scroll or no scroll: no scroll is the default. This tells your cursor how to behave in relation to what your statements say.
3. Cursor holdability is another one. It comes in “with hold” or “without hold”-- “without” hold is the default.
4. The last feature is the cursor returnability. There are also two options: without return (the default) and with return.

With a query expression, you can use any type of SELECT statement. All rows extracted as the result will be turned over to the cursor because they fit

the scope of the cursor. When this is happening the query itself is not executed yet, so data cannot be obtained before the OPEN statement. Once the loop is initiated in the FETCH statement, the data will be analyzed one row at a time.

If there is a specific way you want the data to be processed, you can arrange all the extracted rows before processing by using the “ORDER BY” clause. Here is how the clause looks:

```
ORDER BY sort-specification [, sort-specification]
```

It's very concise, but this does not mean you are limited in the number of ordering instructions you put in. As long the syntax is followed you will have no problems.

```
(column-name) [COLLATE BY collation-name] [ASC|DESC]
```

In most cases, the sorting happens after the name of the column, so it has to be included in the query expression's select list. Columns that are in the table, but are not included in the select list, will not be recognized by ordering instructions. Let's say we want to perform an operation that SQL does not support for chosen rows in a CLIENT table, by declaring:

```
DECLARE cust1 CURSOR FOR  
SELECT CustID, FirstName, LastName, City, State, Phone  
FROM CUSTOMER  
ORDER BY State, LastName, FirstName ;
```

The SELECT statement will extract rows ordered by state, last name, and first name. This way clients from Alaska (AK) will be extracted before those from Alabama (AL). Once that is complete, the clients will be ordered by their last names. If there are identical last names, the sorting will be done after the first name.

If you have ever had the misfortune of printing a ten-page document fifty times without using a copy machine with a collator, you will know that it would involve making ten different paper piles, and then you would have to place each page in the proper pile/sequence. This process is called collation and it takes place in SQL. Collation in SQL is a set of rules that tell us how strings in a character set are compared. For instance, a character set will have a specific collation order which tells it how each element is arranged;

however, you can use a different collation order if you prefer by using the "COLLATE BY " clause.

When declaring a cursor, it is possible to mention a column that is not part of the underlying table. In this case, there isn't a name that can be used with the order by clause, so we will have to name it during the declaration of the cursors. Here is an example:

```
DECLARE revenue CURSOR FOR  
SELECT Model, Units, Price,  
Units * Price AS ExtPrice  
FROM TRANSDetail  
ORDER BY Model, ExtPrice DESC ;
```

Since a collation clause is not specified, the default will be used. Look at the select list and the fourth column. It is the result of data calculation that takes place in two columns preceding it. That column represents extended price and the sorting clause is processed first by following the model name before it deals with the extended price. We have specified in our ORDER BY clause that we want a descending order, so the most valued transactions will be handled first. When you use the clause and you don't specify the order, it will be ordered in ascending order. This is why we used DESC in the sort.

## **Updatability, Sensitivity, and Scrollability**

Often you will find yourself in need of an update to the table row or removal of them. In some cases, you might want to remove the ability of people to make modifications to the table. SQL gives you the ability to do this. All you need are a couple of clauses and statements. Say you want to prevent people from changing the scope of a cursor; you can use the following clause to do that:

**FOR READ ONLY**

If you want to be selective and protect certain columns you can use the following clause:

**FOR UPDATE OF column-name [, column-name]**

All of the columns you list need to be part of the query expression in your cursor declaration. Without an updateability clause, you should know that all

columns will be updatable. In the default case, you can use an “update” statement as you would any other day and update all columns the pointer is pointed towards. The way we can tell that a certain row falls within the scope of a cursor is through the query expressions.

Let’s say you have a statement found between the open and close statements, and all the data is modified. The modifications will remove them from the query’s requirement. What happens next? Will all rows be processed, or will the modified rows be unrecognizable and therefore skipped? Remember that SQL statements like UPDATE and DELETE work with specific rows or the rest of the table. When these are active, other statements, working on the same data, cannot interfere or work at the same time. But when you use a cursor this changes; data can be manipulated by various operations at the same time, making you vulnerable until it is closed. You can see that if you open a cursor and run it, then open another cursor without closing the previous one, everything that happened under the second cursor will affect statements that belong to the second cursor.

So if you modify data under the query expression of a cursor declaration after a number of rows, you will have chaos. You will find yourself with data that can be very misleading or inconsistent. This issue can be avoided by preventing cursors from being changed by other operations or statements. To do this you have to include the “INSENSITIVE” keyword in the cursor declaration. This means when your cursor is open, any modifications made to the table will not change its scope. The cursor can be updatable and insensitive at the same time, but if your cursor is insensitive it has to be read only. Let's look at an example:

```
DECLARE C1 CURSOR FOR SELECT * FROM EMPLOYEE  
ORDER BY Salary ;  
DECLARE C2 CURSOR FOR SELECT * FROM EMPLOYEE  
FOR UPDATE OF Salary ;
```

Say you open two cursors. You get some rows using C1 and update the salary by using C2. The modification can result in a row that is extracted using C1 and then appear again when another extraction is performed by C1. So having multiple cursors open at the same time can lead to strange things. But by isolating your cursors, you avoid problems like this. So avoid working with



multiple cursors at the same time unless it is absolutely necessary.

The sensitivity of a cursor is set to `asensitive` by default. The meaning of `asensitive` changes from implementation to implementation. For instance, it can mean sensitivity and then mean insensitive in another situation. To understand what it means in your situation you will need to read your system's documentation.

Let's now turn to scrollability. Scrollability allows you to change the position of the cursor. This way you can get access to specific rows in a particular order. All you need to do is to add the keyword "SCROLL" when declaring your cursor. You control the cursor's movement within the `FETCH` statement (this will be discussed later). Before exploring this in detail we need to discuss opening cursors.

## Opening the Cursor

As we have discussed, a cursor's declaration statement does nothing. You can include what rows are under the cursors but no action is taken. To have the cursor act you need an `OPEN` statement. An open statement looks like this:

```
OPEN cursor-name;
```

So in order to open the cursor in the "ORDER BY" example we discussed earlier, you would do the following:

```
DECLARE revenue CURSOR FOR  
SELECT Model, Units, Price,  
Units * Price AS ExtPrice  
FROM TRANSDetail  
ORDER BY Model, ExtPrice DESC ;  
OPEN revenue ;
```

Nothing happens until the cursor is opened, as I said previously. The cursor has to be declared to be opened. Let's see how it would look using SQL inside a host language application:

```
EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM ORDERS  
WHERE ORDERS.Customer = :NAME  
AND DueDate < CURRENT_DATE ; NAME := 'Acme Co';  
EXEC SQL OPEN C1; NAME := 'Omega Inc.';  
EXEC SQL UPDATE ORDERS SET DueDate = CURRENT_DATE;
```

(Source: <https://www.dummies.com/programming/sql/how-to-open-a-sql->

cursor/ retrieved in December 2019)

As you can see, the open statement solidifies values that have reference inside the cursor declaration statement. The values of date/time functions are fixed, meaning we have another assignment to the “name” variable, but nothing will be affected in the rows the cursor works. Let’s turn our attention to the fetch syntax.

## Fetching Data

Three events take place when we use cursors. The first one is the declaration of the cursor: this will define the scope and give the cursor a name. Then we use an open statement to initiate activity and gather the row specified in the query expression, Then there is the fetch statement which gathers the information that is required. Let's see how a fetch statement looks:

```
FETCH [[orientation] FROM] cursor-name  
INTO target-specification [, target-specification] ;
```

(Source:

[https://docs.oracle.com/cd/B12037\\_01/appdev.101/b10807/13\\_elems020.htm](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/13_elems020.htm)  
retrieved in December 2019)

It's worth mentioning that you can direct the cursor by using the following orientation clauses: NEXT, PRIOR, FIRST, LAST, ABSOLUTE, RELATIVE, and "simple value specification". If you don't do this the default is "NEXT". In old versions of SQL "NEXT" was the only orientation available.

The "NEXT" clause moves the cursor from its present position to a row that follows as specified in the query expression. So if our query is found before the first record, it will move to the right of it: it will move from record  $x$  to record  $x + 1$ . If it is located at the last record, it will relocate beyond it which might produce a "no data" notification.

If a module language or embedded SQL is using the cursor, target specification will be parameters or host variables. The type and number of specifications will have to be the same as the name of columns, just as types, that are in the query expression in the cursor declaration. For instance, if we want to retrieve ten item values from the table row, we will need to have ten matching host variables that will accept those values.

We talked earlier about the scalability of cursors and that there are other options to the default settings, the most obvious being the PRIOR option which makes the cursor relocate itself to a row right before the present location. If we choose "FIRST" the cursor will go to the first record; the "LAST" option does the opposite. When you use the RELATIVE or

ABSOLUTE options, you need an integer present. So if we declare “FETCH ABSOLUTE 20” the cursor will move to the 20th row. The “RELATIVE” options will relocate to the 20th row after the present location is on. You should know that if you declare RELATIVE 0, the cursor will not move. RELATIVE 1 will act like next and RELATIVE -1 will be equal to PRIOR.

We said that update and delete operations can be performed on a row the cursor is pointing to. Since there is no row after the pointer, you will get an error instead of deletion. Here’s how these operations would look:

```
DELETE FROM table-name WHERE CURRENT OF cursor-name ;  
UPDATE table-name SET column-name = value [,column-name = value]  
WHERE CURRENT OF cursor-name ;
```

Remember that an update will be made when the value is equal to the expression value or the DEFAULT. If you get an error the process will be terminated.

When you are done with the cursor, remember to close it or it will not stop running. You may encounter problems if other operations run while the cursor is open.

You would do so by typing:

```
CLOSE revenue ;
```

# CHAPTER 5:

## **Procedural Capabilities**

Creating a database standard that could be followed by everyone was one of the biggest goals in the world, and we have achieved that. Once that was achieved, work continued to make our system more efficient, adding more functions. In other words, we were building a new standard. In the gap between the first standard and the new one, models were introduced to SQL that added a series of statements that allowed SQL to work with control flow, like in programming languages like C. SQL can now be used by programmers to perform a lot of operations that were only available in other programming languages.

In this chapter, we will explore those SQL improvements: the compound statements, control statements, and much more.

## **Compound Statements**

So far we have talked about SQL as a non-procedural programming language that works with data instances in arranged sets instead of dealing with it record by record. We are now going to learn that SQL is a far more sophisticated language than was originally let on. We will see that SQL is shifting to a more procedural language.

SQL 9 did not have any kind of procedural model; there were no instruction sequences. Earlier versions of SQL worked with standalone statements combined with C++ and Visual Basic software. In those days, executing a series of statements was not recommended because of what systems can

handle and low networking speeds. All of this changed with the introduction of compound statements in 1999, allowing separate statements to work as a single statement, resolving network worries in the process. Statements that make up compound statements need to be put between two keywords: “BEGIN” and “END”. Here is how that would look:

```
void main {  
EXEC SQL  
BEGIN  
INSERT INTO students (StudentID, Fname, Lname)  
VALUES (:sid, :sfname, :slname) ;  
INSERT INTO roster (ClassID, Class, StudentID)  
VALUES (:cid, :cname, :sid) ;  
INSERT INTO receivable (StudentID, Class, Fee)  
VALUES (:sid, :cname, :cfee)  
END ; /* Check SQLSTATE for errors */ }
```

(Source: <http://pubhtml5.com/vmmz/vhzn/basic/501-550> retrieved in December 2019)

The example above is written with C with compound SQL inside. The last comment deals with errors. If the code does not run as it should we will get an error message in the SQLSTATE parameter. I have put it there because it is a best practice. This is something you should do as it is helpful; you will avoid a lot of unnecessary strife this way.

There is a risk that when you work with compound statements you might encounter errors you have never experienced before. To make things worse, compound statements don’t act like common SQL statements when things go wrong. For instance, common statements will not alter the database when something goes wrong. But compound statements do not do that. Looking at the example above, consider the two INSERT statements to the STUDENTS and ROSTER tables. What happens if both statements are executed at the same time? Because of the involvement of the second user, the INSERT statement would fail to go into the RECEIVABLE table. This means a student will be registered without being taxed; the university would have to cover that somehow.

This is atomicity in action. Atomic statements are either fulfilled or they aren’t. There are no ifs and buts about it. This is how common SQL

statements function; they are atomic. Compound statements, on the other hand, are not atomic. This can be remedied by following the syntax below:

```
void main {  
EXEC SQL  
BEGIN ATOMIC  
INSERT INTO students (StudentID, Fname, Lname)  
VALUES (:sid, :sfname, :sname) ;  
INSERT INTO roster (ClassID, Class, StudentID)  
VALUES (:cid, :cname, :sid) ;  
INSERT INTO receivable (StudentID, Class, Fee)  
VALUES (:sid, :cname, :cfec)  
END ; }
```

(Source:

[https://docs.oracle.com/cd/A64702\\_01/doc/server.805/a58232/ch05.htm](https://docs.oracle.com/cd/A64702_01/doc/server.805/a58232/ch05.htm)  
retrieved in December 2019)

As you may have noticed, the ATOMIC keyword is introduced just next to the BEGIN keyword. To ensure that the execution of the state is how we desire it, you can put the keyword before the BEGIN keyword. When our statements encounter an error the statement will revert everything to the way it was. In other words, nothing will be changed.

A common feature of full programming languages is the use of variables. SQL didn't have a variable until SQL/PSM. A variable is essentially a symbol that acts as a container for a value (this value comes in different data types). What is interesting about compound statements is that variables can be declared inside them, received, and used. This feature is very useful, since as you might know, variables are like the building blocks of many programming functions. Once the statement is finished, the variable will be erased. Let's take a look at an example:

```
BEGIN  
DECLARE ceopay NUMERIC ;  
SELECT salary  
INTO ceopay FROM EMPLOYEE  
WHERE function = 'ceo' ; END;
```

A cursor can also be declared inside a compound statement. As we have alluded to, the cursor can then be used to compute information row after row. In this case, you needn't worry about the cursor remaining open until you

close it. It has the same shelf life as the variable because it gets terminated when the compound statement is finished executing. Here is an example below:

```
BEGIN DECLARE potential_employee CHARACTER(30);  
DECLARE c1 CURSOR FOR  
SELECT business  
FROM techytech;  
OPEN CURSOR1;  
FETCH c1 INTO potential_employee;  
CLOSE c1; END;
```



## Managing Conditions

SQL statements give us one of the following results whenever they are executed: a successful result, an ambiguous result, or a completely faulty result. All these meet a specific condition or are brought about by one. When an SQL statement is being executed, the database server will assign a value to the SQLSTATE parameters which will show whether a statement is executed correctly. If there is an error the parameter will give us information about it. The SQLSTATE parameter has a field of five characters, and the first two characters will tell us if the statement was successful. Here's how those codes look:

1. Successful Completion (class code 00): This statement is self-descriptive. You get it when the statement is executed successfully.
2. Warning (class code 01): This message means something strange has occurred, but the database message system can't classify it as an error or something else. In other words, it prompts you to go check your SQL statement to make sure that everything is as intended. There might be a factor you are overlooking.
3. Not Found (class code 02): When this happens you might receive a message that says "no data was returned". This isn't necessarily a bad thing; sometimes what you are looking for isn't there or you are hoping to find nothing. It depends a lot on your situation,
4. Exception (other class codes): When you receive this error the message will include other details about the error that tell you what it is, and from there you can figure out how to tackle it.

The first code requires no actions, but class codes 01 and 02 require you to take some action. If you are working on something and you expect a "not found result" or a warning message, you will probably do nothing because that falls within what you expected. If you were not expecting any of this, action is required. This is why as a developer you will have to create an execution procedure to handle such situations that are hard to anticipate. All exception handling producers will depend on the type of error faced. You should not have an exception handling procedure that handles multiple

exceptions because each will demand a different solution. The obvious options with dealing with errors is working around them or fixing the error. I like the idea of choosing how to act after understanding the error. In some cases, you might be dealing with a fatal error that kills your programs.

You can put condition handlers in your compound statements. To do this you have to declare the conditions which the handler will deal with (it can be an exception or a specific error code). Let's take a look at this:

```
BEGIN DECLARE constraint_violation CONDITION  
FOR SQLSTATE VALUE '23000' ; END ;]
```

When the condition is met the handler will be executed. The handler will have a specific task when the condition is met. It can be something as simple as an SQL statement or compound statement. When that statement has executed the “effect” of the exception handler will be executed. Here is a list of exception handler effects:

1. CONTINUE: This means when the exception handler is called the execution of the statement will continue.
2. EXIT: The execution will continue only after the compound statement that holds the exception handler is triggered.
3. UNDO: In this case, any action caused by past statements within the compound statement will be reversed to the previous state. The execution will then continue after passing the statement that holds the exception handler.

If the exception handler is expected to resolve the problems that triggered it, it's best to choose the CONTINUE effect. The EXIT effect can work in that situation too but it is much better in situations where the exception handler can't fix the problem and you don't need changes to be reversed. The UNDO effect is best if you prefer to revert everything to the way it was before you ran a statement. Let's look at an example:

```
BEGIN ATOMIC  
DECLARE constraint_violation CONDITION FOR SQLSTATE VALUE '23000' ;  
DECLARE UNDO HANDLER  
FOR constraint_violation RESIGNAL ;  
INSERT INTO students (StudentID, Fname, Lname)  
VALUES (:sid, :sfname, :slname) ;  
INSERT INTO roster (ClassID, Class, StudentID)
```

```
VALUES (:cid, :cname, :sid) ; END ;
```

(Source: <https://www.dummies.com/programming/sql/how-to-handle-sql-conditions/> retrieved in December 2019)

You should know that when a constraint violation is caused by an INSERT statement you will get a 23000 result. This can happen when there is a new record with a key identical to one that already exists. All modifications that happened after the INSERT command will be undone through the UNDO effect.

The RESIGNAL statement will turn over control to the current procedure. If our INSERT statements come across issues, the execution will continue as normal. In the example, we are using the ATOMIC keyword. It is obligatory due to the UNDO effect found under the exception handler. Also, you do not need to use the keyword if you are using EXIT or CONTINUE effects.

Let's look at a different scenario. The exception value is something other than 23000, and the exception handler doesn't know how to fix the error thrown. What are we to do? In an instance like this, RESIGNAL will be triggered which means that the problem will move to the next control where it can be handled. If a solution is not found, it is pushed again to the next one. This illustrates that what you need are more contingencies. So if you are writing a statement that might return an exception, you need to have handlers for all the possibilities. You will take care of what you cannot predict.

# The Flow of Control Statements

One of the biggest factors that discouraged people from working with SQL procedurally was because SQL didn't have a flow of statements. Before SQL/PSM, there wasn't a way of getting out of the restrictive execution sequence without the help of C. SQL/PSM provided the control structure flow that was offered by other languages. Today, SQL can perform various functions without accessing other programming languages.

Let's start with the most basic statement: "IF THEN ELSE END IF". If you have worked with other programming languages, you will know what this statement means. If a certain condition is met, statements after the "then" keyword will be executed. If something else happens, those after the "else" keyword will be executed." Here is an example:

```
IF exname = 'Sam' THEN
UPDATE students
SET Firstname = 'Sam'
WHERE StudentID = 23343 ;
ELSE DELETE FROM students
WHERE StudentID = 23343 ;
END IF
```

In the example, we have an "exname" variable and if it contains the value "Sam", the particular record will be updated. If the variable is equal to another name it will be removed from the table. This is the sort of thing you will do in a fully-featured programming language. You can already see how logic like this can be useful, as there are many circumstances where something similar is needed. You may need to change something in the database that meets specific criteria and skip other values that don't meet those criteria, for example.

A more rigorous statement than the "if-else" statement is the Case statement. We have the basic case statement and then we have the searched statement. Both enable us to utilize various execution paths that depend on the condition's value. The basic statement checks one condition and depending on its value, an execution will take a particular path. Here's how it looks:

```
CASE mymajor
WHEN 'Computer Engineering'
```

```
THEN INSERT INTO nerds (StudentID, Firstname, Lastname)
VALUES (:sid, :sfirstname, :slastname) ;
WHEN 'Athletic Training'
THEN INSERT INTO athletes (StudentID, Firstname, Lastname)
VALUES (:sid, :sfirstname, :slastname) ;
WHEN 'History'
THEN INSERT INTO historians (StudentID, Firstname, Lastname)
VALUES (:sid, :sfirstname, :slastname) ;
ELSE INSERT INTO undecided (StudentID, Firstname, Lastname)
VALUES (:sid, :sfirstname, :slastname) ;
END CASE
```

We have added the ELSE clause so that everything that isn't included in the categories we declared in the THEN clause can be caught. The ELSE clause is optional; use it when you think it is appropriate for your situation. If none of your clauses can handle one of the conditions and you have no ELSE statement, an SQL exception will be thrown.

Then we have the LOOP statement. The LOOP statements allow us to run SQL statements x number of times. When the last line of code is executed, an ENDLOOP statement will go back to the first line and run the code again, Here's how it looks:

```
SET mycount = 0 ; LOOP
SET mycount = mycount + 1 ;
INSERT INTO rock (RockID)
VALUES (mycount) ;
END LOOP
```

The code will preload the "rock" table with unique identifiers. All the different rocks and their details will be presented as they are found. The one drawback in this code is that it is never-ending, so table rows will be inserted until we run out of space. When this happens the database management system might throw an exception, but in some cases, the system might crash. A loop is only useful when we exit it as soon as the job is done. To do this we need to add a LEAVE statement. When the execution meets the line it breaks the loop and switches to another task.

Let's now take a look at the WHILE statement. The WHILE statement will repeat a statement as long as certain conditions are true. When the conditions no longer apply, the while statement stops.

If you have worked with any fully featured programming language you will know how a WHILE statement works. You might want to run a set of statements that find a number in an unknown number of values and update them. In such a situation you need to identify those values and update them as long as they remain unchanged with the WHILE statement. Here is how it looks:

RockPreparation:

```
SET mycount = 0 ;  
WHILE mycount < 10000 DO  
SET mycount = mycount + 1 ;  
INSERT INTO rock (RockID)  
VALUES (mycount) ;  
END WHILE RockPreparation
```

This example works in the same manner as the example we used previously. This is to illustrate that it can work in different ways. This is how flexible SQL has become.

The final loop type is the FOR loop. If you have worked with C#, the FOR loop works in a similar way as it does there. It can be used to declare and open a cursor, then gather all rows and execute statements within the FOR statements. Once the work is completed the cursor will be closed. It is a very useful loop if you'd rather just rely on SQL in your build. As you might know now, this is one of the most useful loops in any programming language. Here's how it looks in SQL:

```
FOR mycount AS C1 CURSOR FOR  
SELECT RockID FROM rock  
DO UPDATE rock SET Description = 'shiny onyx'  
WHERE CURRENT OF C1 ;  
END FOR
```

With this loop, all rows inside the ROCK table will be updated with a “shiny onyx” description. It is an effortless way of adding descriptions to what may be a long list of rocks. In a real work setting, you might have to tweak it so that it updates accurately (you wouldn't want every rock to have the same description).

The ITERATE statement allows us to modify execution flow. Iterated statements are the loops we have talked about: the LOOP, WHILE, FOR, and

REPEAT. When the condition of an iteration statement is true, or absent, the next loop iterations will take place after the ITERATE statement. If the condition is false, the next iteration after the ITERATE STATEMENT will not execute. Let's see how it looks:

Rock Preparation:

```
SET mycount = 0 ;  
WHILE mycount < 10000 DO  
SET mycount = mycount + 1 ;  
INSERT INTO rock (RockID)  
VALUES (mycount) ;  
ITERATE RockPreparation ;  
SET mypreparation = 'FINISHED' ;  
END WHILE RockPreparation
```

## Stored Procedures and Functions

Stored procedures are unique because they are a part of the server instead of the executing client. They are important because they allow for minimum network traffic while improving performance. So the only traffic that is of concern to the developer is that which comes from the client to the server. This is a CALL statement that is used to invoke a stored procedure once it is defined. Let's have a look at the procedure:

```
EXEC SQL  
CREATE PROCEDURE ChessMatchScore  
( IN score CHAR (3),  
OUT result CHAR (10) )  
BEGIN ATOMIC  
CASE score  
WHEN '1-0' THEN  
SET result = 'whitewins' ;  
WHEN '0-1' THEN  
SET result = 'blackwins' ;  
ELSE  
SET result = 'draw' ;  
END CASE END ;
```

(Source: <https://www.dummies.com/programming/sql/how-to-use-sql-stored-procedures/> retrieved in December 2019)

Now, this is the CALL statement that invokes the procedure once it is created:

```
CALL ChessMatchScore ('1-0', :Outcome) ;
```

An input parameter is inserted into the procedure as the first argument. The second argument is an embedded variable that takes in a value designated to the output parameter which returns the result.

Stored procedures, like most technologies, have improved over the years. One of the most important improvements is the addition of named arguments. Here is how the previous statement looks with named arguments:

```
CALL ChessMatchScore (result => :Outcome, score => '1-0');
```

Named arguments can always be put in sequential order. Wherever their position, we cannot confuse them.

Another improvement is the introduction of default input arguments that can be declared of the input parameter. Once you do this, there's no need to declare the input value inside the call statements because defaults are assumed.

Stored functions are almost identical to stored procedures. They are both stored routines but are not invoked the same way. While a stored procedure is invoked by a CALL statement, stored functions are called with a function call. The function call can replace or stand for an SQL statement. Here's how it looks:

```
CREATE FUNCTION PurchaseHistory (CustID) RETURNS CHAR VARYING (200)
```

```
BEGIN
DECLARE purch CHAR VARYING (200)
DEFAULT "";
FOR x AS SELECT *
FROM transactions t
WHERE t.customerID = CustID DO      IF a <> ""
THEN SET purch = purch || ' ' ;
END IF ;
SET purch = purch || t.description ;
END FOR
RETURN purch ; END ;
SET customerID = 314259 ;
UPDATE customer
SET history = PurchaseHistory (customerID)
WHERE customerID = 314259 ;
```

(Source: <https://www.dummies.com/programming/sql/how-to-use-sql-stored->



procedures/ retrieved in December 2019)

This function will generate a list of purchases made by the customer. The customer's ID number is used to extract information from the transaction tables. The update statement calls the purchase history to add the customer's most recent purchases.

# CHAPTER 6:

## Collections

In most programming languages the word array refers to a collection of items. In Oracle, arrays are known as collections that contain a list of elements that belong to a similar type. Elements that are in an array/collection can be accessed by their index. Each spot taken has its own index. The difference with collections is that they are more useful and they are a standard feature in most database programs.

Collections can boost an application's performance and they also provide data caching tools that can really improve an application. A collection can be a database column, a subprogram parameter, or an attribute. In this chapter, we are going to focus on all those features of collections.

### Overview

Collections are essentially one-dimensional structures that contain an ordered list of elements. The architecture of each element is based on cells that have subscripts. The elements inhabit cells and we can access them by their index. The subscript and the index are the same thing; their use is to help us access an element and do something with it.

SQL type data, and others, can be items in a collection. For instance, an SQL primitive is a scalar value, and the user-defined type is the object type. Collections can be used in PL/SQL programs if we declare PL/SQL variable as a collection type. The incredible thing is that the variable can also contain an instance of its collection type. What's more, database collections can

belong to the collection type structure. Since collections are one-dimensional structures you need to keep in mind that you cannot use two-dimensional coordinates on them. The way multidimensional arrays can be created is if the collection contains an object types attribute. There are two categories of collections: bounded and unbounded. Bounded collections hold a specific number of elements amidst a host of other restrictions. Unbounded collections are freer and their numbers can go as high as the system will allow.

Collections can also allow us to order data inside an array while working with object-oriented features at the same time. For instance, a nested table can be accessed while its information is being deposited in the table columns. We can use collections to take advantage of data caching and improve our applications' performance which will improve the operations of our SQL.

When working with dedicated server connections, we will need to make use of PL/SQL elements called the "User Global Area" to perform various collections operations. In a shared server mode these operations can be handled outside the user global area. The user global area is also a component of the system global area. This means we can have several server processes affecting the session, which puts the user global area outside the system global area.

Another important aspect of collections is that they can be persistent or nonpersistent. A collection is persistent when it deposits its structure and its elements inside the database. A non-persistent collection only runs for a single session while a particular application runs.

On top of all this, collections come in three different formats: associative arrays, nested tables, and varrays. These formats are dependent on the characteristics and attributes of the PL/SQL application. Let's take a look at these formats:

1. **Associative arrays** are also known as "index by" tables. They are the most basic type of collection and they are non-persistent and unbounded. This means they cannot store data inside the database, however, we can access it inside the PL/SQL block. After the application runs, the structure and the data of the array will be lost.

The reason why it is called the index table is that we can use indexes to identify elements in the array.

2. A **nested table** is persistent and unbounded. They are created inside the database, and we can also access them in the PL/SQL block.
3. **Variable size arrays**, also called varrays, are persistent and bounded. We can set them up in a PL/SQL block or the database. They have a one-dimensional structure. The only difference between varrays and nested tables is their limited size and nested tables are not limited.

Now, we should discuss the best use scenarios for each of these collection types:

1. **Associative arrays** are best when you need to cache application data temporarily to use later during a process. It is also desirable if you want string subscripts for elements in a collection. They also work best when we need to map hash tables from the client to the database.
2. **Nested tables** work best when we want to store a set of information directly into the database. The columns in a nested table will need to be declared specifically so they can contain data in a persistent manner. Nested tables are used to perform powerful array operations and handle big sets of data.
3. **Varrays** are best used to store a predetermined collection in a database since it offers you a limited space to store your information. They are also best used in preserving the order of a collection of elements.

Now that you have some idea of what they are and how they function, let us look at them in more detail.

## Associative Arrays

As mentioned, associative arrays are like lists inside PL/SQL applications, and their structure and data cannot be put inside the database. They are also not limited in how much they can store and they can contain various types of elements in a key/value architecture.

Associative arrays were introduced in PL/SQL tables to be used in PL/SQL blocks. Oracle 8 was the first to identify PL/SQL tables as an associative array because of the index/value structure it had. In Oracle 10 the associative array received its official name because of the array-like usage of indexes. Let's look at the syntax of arrays:

```
TYPE [COLL NAME] IS TABLE OF [ELEMENT DATA TYPE] NOT NULL  
INDEX BY [INDEX DATA TYPE]
```

In the example, the index type is the data type of the subscript. Only the following index data types are not supported: NUMBER, RAW, ROWID, CHAR, and LONG-RAW. The following index data types are supported: PLS\_INTEGER, BINARY\_INTEGER, NATURAL, POSITIVE, SIGNTYPE, and VARCHAR2. Data types of each element can belong in the following categories:

1. **Scalar data types:** Here is where we find the NUMBER data type with all its subtypes and others like VARCHAR3, BOOLEAN, BLOB, CLOB, AND DATE.
2. **Inferred data** are inferred data types from the table column or cursor expression.
3. **User-defined data** types are user-defined collection types or data item types.

Now, let us look at the PL/SQL program where an associative array is declared inside the block. You will see that the array's subscript belongs to the string type and it deposits a number of days within a quarter. I put this here to show you how to declare an associative array and assign each element inside a cell and print them.

```
/* The SERVEROUTPUT is enabled in order to display the output*/  
SET SERVEROUTPUT ON  
/*Run the PL/SQL block*/  
DECLARE  
/*Declare a collection type associative array and its variable*/  
TYPE string_asc_arr_t IS TABLE OF NUMBER  
INDEX BY VARCHAR2(10);  
l_str string_asc_arr_t;  
l_idx VARCHAR2(50);  
BEGIN  
/*Assign the total count of days in each quarter against each cell*/
```

```

l_str ('JAN-MAR') := 90;
l_str ('APR-JUN') := 91;
l_str ('JUL-SEP') := 92;
l_str ('OCT-DEC') := 93;
l_idx := l_str.FIRST;
WHILE (l_idx IS NOT NULL)
LOOP DBMS_OUTPUT.PUT_LINE('Value at index '||l_idx||' is '||l_str(l_idx));
l_idx := l_str.NEXT(l_idx);
END LOOP; END; /
Value at index APR-JUN is 91
Value at index JAN-MAR is 90
Value at index JUL-SEP is 92
Value at index OCT-DEC is 93

```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

Look carefully at the string indexed array in the example. This will improve performance by introducing an indexed array of values. In the final block we see clearly how data is assigned.

Now let us have another application that is going to fill an array automatically by itself. We are going to declare the associative array as containing all ASCII number values from 1 to 100.

```

/*The SERVEROUTPUT is enabled in order to display the output*/
SET SERVEROUTPUT ON
/*Start the PL/SQL Block*/
DECLARE
/*Declare an array of string indexed by numeric subscripts*/
TYPE ASCII_VALUE_T IS TABLE OF VARCHAR2(12)
INDEX BY PLS_INTEGER;
L_GET_ASCII ASCII_VALUE_T; BEGIN
/*Insert the values through a FOR loop*/
FOR I IN 1..100 LOOP
L_GET_ASCII(I) := ASCII(I);
END LOOP;
/*Display the values randomly*/
DBMS_OUTPUT.PUT_LINE(L_GET_ASCII(5));
DBMS_OUTPUT.PUT_LINE(L_GET_ASCII(15));
DBMS_OUTPUT.PUT_LINE(L_GET_ASCII(75));
END; /
53 49 55

```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

From these, we can see that associative arrays can be scattered or empty collections. But, they cannot be part of DML transactions because they are non-persistent. We also see that we can use the same block to pass arguments to other subprograms inside. Associative arrays can also be declared inside package specifications to make them act like persistent arrays.

## **Nested Tables**

As I have said, a nested table is a persistent collection that is built inside the database and can be accessed in the PL/SQL block. We said they are unbounded collections that rely on indexes and depend on a server whenever information is extracted. Oracle will label the minimum subscript with 1 and then relatively deals with the rest. Nested arrays don't follow the index/value architecture we saw earlier because they can be accessed the same way as an array within a block. Nested tables are generally dense, but they can evolve into sparse collections, especially when we have various delete operations happening over the cells. A dense collection, as the name suggests, means there are few or no empty cells between indexes. Sparse collections have more empty spaces in between indexes.

When we declare a nested table inside a PL/SQL application, it will work the same way as a one-dimensional array that doesn't have an index type or a limit. If nested tables are declared within a database, they will exist as a valid schema data item type. We can use nested tables inside PL/SQL blocks to declare variables that will hold data temporarily, or we can use them in a database where they will be inserted in a table that will permanently store data inside them. Nested table columns are similar to tables within tables, but it is a table that contains nested table information.

When we create a nested table inside a database column inside another table (known as parent table), Oracle will make a table for storage with options identical to those in the parent table. This storage table will have a name that is inside the "STORE AS" clause of the nested table. You also need to know that Oracles servers perform two actions when a new row is created in the parent table:

1. First, a new identifier is created in order to differentiate instances

of various parent rows

2. The nested table is generated inside the storage table with a new identifier.

All nested table operations and processes will be handled by the Oracle server. Programmers don't see this happening because it is hidden under the insert-type operation. Here is how the syntax of a nested table in PL/SQL looks:

```
DECLARE TYPE type_name IS TABLE OF element_type [NOT NULL];
```

In the example above, the element type is a primitive data type, but it can also be a user defined data type. Here's how the syntax of a nested table inside the database looks:

```
CREATE [OR REPLACE] TYPE type_name IS TABLE OF [element_type] [NOT NULL];
```

We have an element type, but it is either a scalar data type that is supported by SQL, a REF item, or a database type. Remember that all non-SQL data types are not on the supported element type list; these are data types like NATURAL, LONG, BOOLEAN, REF CURSOR, STRING, and so on. If we want to change the element type size of the database collection type we can do the following:

```
ALTER TYPE [type name] MODIFY ELEMENT TYPE [modified element type] [CASCADE | INVALIDATE];
```

CASCADE and INVALIDATE will determine whether the dependents need to be invalidated by the collection modification.

Then we can drop the statement by using the DROP statement. We can use the DROP statement in conjunction with the FORCE keyword to force it to drop regardless of any dependents attached to the statement. Here is how it looks:

```
DROP TYPE [collection name] [FORCE]
```

Since the collection type is created within the database, we can use it to declare columns. To do that we use the CREATE TABLE command and declare a column of the nested table. Inside is the parent table. Let's see how that gets put together:

```
SQL> CREATE TABLE TAB_USE_NT_COL
```



```
(ID NUMBER, NUM NUM_NEST_T)
NESTED TABLE NUM STORE AS NESTED_NUM_ID;
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

Now, let's use a collection constructor to add the nested table information. The constructor is Oracle's, and it generates a value for every single attribute.

```
INSERT INTO TAB_USE_NT_COL (ID, NUM) VALUES (1, NUM_NEST_T(10,12,3)); /
1 row created.
INSERT INTO TAB_USE_NT_COL (ID, NUM) VALUES (2, NUM_NEST_T(23,43)); /
1 row created.
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

Now let us look at how you select a nested table column. Notice that when the column is queried, the nested column is presented as one of the instances of the nested table item type.

```
SQL> SELECT *
FROM tab_use_nt_col;
ID NUM
1 NUM_NEST_T(10, 12, 3)
2 NUM_NEST_T(23, 43)
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

We can use the TABLE statement to initiate an instance and display all the data in the form of relational data. With the statement, we will be able to gain access to any attributes inside the nested table type. Because of this, Oracle will combine the parent row with the nested row when it reaches the output of the query.

Let's look at how we update nested table instances. There are few ways we can do this: we can cut and paste a TABLE statement, or we can replace the instance with another by using an UPDATE statement.

```
UPDATE tab_use_nt_col SET num = num_nest_t(10,12,13) WHERE id=2 /
1 row updated.
SQL> SELECT * FROM tab_use_nt_col;
ID NUM
1 NUM_NEST_T(10, 12, 3)
```

```
2 NUM_NEST_T(10, 12, 13)
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

Notice that by using the TABLE expression it is possible to update just one collection item. Here's how that would look:

```
UPDATE TABLE (SELECT num FROM tab_use_nt_col WHERE id = 1)
P SET P.COLUMN_VALUE = 100 WHERE P.COLUMN_VALUE = 12;
```

As you can see, we have an UPDATE statement where a TABLE expression examines the instance of the collection type in which the subquery returns. As a result, the instance is opened in a relational format, allowing us to have access to the values and make modifications to them. The subquery only needs to return one instance and one row.

Now let's turn our attention to nested table collections in PL/SQL. In PL/SQL we can declare nested tables in the declaration segment of the block, which is in the form of a local collection type. When the nested table conforms to the item orientation, the PL/SQL variable is initialized. If this does not happen, the Oracle server will throw an exception because the nested table type is not initialized when the block is executed. With the declaration of the nested table collection type, the scope represents the execution block.

Below we are going to declare a nested table in a PL/SQL block. Look at the variable's scope and visibility. The COUNT method will be used to show us all the elements of the array.

```
/*The SERVEROUTPUT is enabled in order to display the output*/
SET SERVEROUTPUT ON
/*Run the PL/SQL block*/
DECLARE
/*Declare a local nested table collection type*/
TYPE LOC_NUM_NEST_T IS TABLE OF NUMBER;
L_LOCAL_NT LOC_NUM_NEST_T := LOC_NUM_NEST_T (10,20,30);
BEGIN
/*A FOR loop is used in order to parse the array and print all the elements*/
FOR I IN 1..L_LOCAL_NT.COUNT LOOP
DBMS_OUTPUT.PUT_LINE('Printing '||i||' element: '||L_LOCAL_NT(I));
END LOOP;
END; /
```

Printing 1 element: 10 Printing 2 element: 20 Printing 3 element: 30

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

That's perfect! It works like a charm.

# Varrays

Varrays are added to Oracle as altered nested tables. Varrays are persistent but bounded, so it makes sense that this is the way that Oracle will perceive them as a version of nested tables. When a varray is declared, a restriction of the elements is also declared. We have the minimum bound which is equal to index 1, the current bound which is the total number of current elements, and the maximum bound which is the maximum number of elements the array can hold. The current bound can never surpass the maximum bound, and the current bound is informative—it tells us how many elements there are and is not prescriptive. Varrays, as I have alluded to, can be built in the form of database items. They can also be used in PL/SQL. All of this is why we have been saying varrays are similar to nested tables but with slight modification storage-wise.

Varrays are the same as the parent record, as a raw value that exists within the parent table. This storage mechanism nullifies the need for a declaration of a storage clause, and we don't need to specify a unique identifier, too. There are some exceptions when dealing with huge amounts of data; for instance, Oracle will store the varray in the form of a LOB. The purpose of the inline storage system is to minimize the total number of IO's on the disk to boost performance and offer better results even in cases where nested tables are concerned.

Varrays can also work as valid table column types and as object type attributes because they are database collection types. Furthermore, when we declare a varray inside a PL/SQL block, it becomes visible on this block alone. Now, let's see how varrays are defined in database collection types:

```
CREATE [OR REPLACE] TYPE type_name IS  
{VARRAY | VARYING ARRAY} (size_limit)  
OF element_type In PL/SQL  
DECLARE TYPE type_name IS {VARRAY | VARYING ARRAY}  
(size_limit) OF element_type [NOT NULL];
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

You will notice that the code includes a size limit that tells us the number of

items that can be part of the array. If for any reason you find yourself needing to change the size of the array, you can do so with the following syntax:

```
ALTER TYPE [varray name] MODIFY LIMIT [new size_limit] [INVALIDATE | CASCADE];
```

This is similar to what we did when we were working with nested arrays. `INVALIDATE` and `CASCADE` function work the same way as they do with nested arrays. Like we did with nested tables, we can also make use of the `DROP` command:

```
DROP TYPE [varray type name] [FORCE]
```

Because of these similarities, we can declare varrays to the PL/SQL block and it follows the same object orientation. For this reason, varrays will always need initialization before they are accessed in the block's execution.

```
/*Enable the SERVEROUTPUT to display the results*/
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE
/*Declare a local varray type, define collection variable and initialize it*/
TYPE V_COLL_DEMO IS VARRAY(4) OF VARCHAR2(100);
L_LOCAL_COLL V_COLL_DEMO := V_COLL_DEMO('Oracle 9i',
'Oracle 10g', 'Oracle 11g');
BEGIN
/*Use FOR loop to parse the array variable and print the elements*/
FOR I IN 1 L_LOCAL_COLL.COUNT LOOP
DBMS_OUTPUT.PUT_LINE('Printing Oracle version:' ||L_LOCAL_COLL(I));
END LOOP; END; /
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

Let's take look at how a varray is built in the form of a database collection type with `SELECT` and DML operations:

```
SQL> CREATE OR REPLACE TYPE num_varray_t AS VARRAY (5) OF NUMBER; /
```

All information about the varray is stored by Oracle inside dictionary views. Now, let's set up a new table that contains a varray type column (we will not use the "store as" clause).

```
CREATE TABLE tab_use_va_col (ID NUMBER, NUM num_varray_t);
/*Query the USER_VARRAYS to list varray information*/
SELECT parent_table_column, type_name, return_type, storage_spec
```

```
FROM user_varrays WHERE parent_table_name='TAB_USE_VA_COL' /  
PARENT_TAB TYPE_NAME RETURN_TYPE STORAGE_SPEC  
NUM NUM_VARRAY_T  
VALUE DEFAULT
```

(Source: <https://hub.packtpub.com/plsql-using-collections/> retrieved in December 2019)

I believe with all we have covered here, you have enough information to deal with collections, handle database transactions, and work with PL/SQL, so now we can move forward and discuss advanced interface methods.

# CHAPTER 7:

## Advanced Interface Methods

Oracle has become the most flexible database manager because it is compatible with a large array of application clients. External routines allow for communication between applications and the database because these other languages still have something to add. There are many cases where we need to call a non-PL/SQL program from PL/SQL. To solve this, Oracle offers external routines. In this chapter, we are going to focus on external routines and how to execute external applications from PL/SQL.

### External Routines

Like I said, external routines allow us to run applications that were written in another language to be used in the PL/SQL environment. Through them, we can work with Java, C, C++ or another application by invoking it with PL/SQL. Such applications are called external programs. The most common place we see this is in program development environments that adhere to the client's uniformity and that of the API. As soon as the external routine is published we can work with it through PL/SQL.

Before diving into the methodology, we need to take a look at a few structural components of Oracle which support external routines:

1. The `extproc` method represents the entire architecture. It is the process that receives requests from Oracle listeners and then runs the external program. The method loads all DLLs, processes arguments it receives from running the program, and transmits it

back. It is the most important component in the structure.

2. The shared library is the way all external programs need to be executed inside PL/SQL. In Windows-based systems, the shared library will present itself as DLLs; in Unix and Linux systems the files will have the .so extension. The language that the library uses has to allow the shared library to be called from languages like C, C++, Visual Basic, COBOL, and FORTRAN. Inside the libraries can exist other programs that can be called in the same manner, so we can load fewer libraries and access many programs.

As you know, dynamic linked shared libraries are easy to generate and they are easy to maintain when compared to statically linked elements. In this case, the shared libraries will be automatically loaded with the extproc method and external programs found there will be executed right away. The result will then be sent to the extproc process which then sends it to the PL/SQL call process.

When working with external routines you need to keep two things in mind: callouts and callbacks. When an external procedure is called we name this the callout. A callback is when a statement is invoked by an external procedure to run the database engine; this statement can either be SQL or PL/SQL that will appear as an operation to the Oracle server. The structure of processing components depends heavily on callouts and callbacks.

When a request from an externally performed action is received by the PL/SQL runtime engine, the call is directed to Oracle common language runtime connection data (the default TNS service for all external routine connections). When the database application is installed, all these connections are configured and managed by the Net configuration assistant. So, the external routine connections and the data server can communicate with each other.

The Oracle common language runtime, ORACLR, is a host whose purpose is to manage external processes that are used to call all non-PL/SQL applications in PL/SQL. But we can make changes to the default configurations to improve security. The process works by receiving a key from the TNS service address and, together with active listeners, they check



the network. This is also how the extproc process is invoked, by the way. Remember that this process requires a DLL path so that we load a shared library. Also, all transfers of the external program are handled with the input arguments, if any are found.

## The Net Configurations

As we have noted the extproc process runs between the external programs and PL/SQL for communications. This process is invoked by a listener and collaborates with external applications through shared libraries to send the results back the same route. What I didn't say is that the activation of the extproc process is managed by net services. One of these services is TSNAMES and LISTENER FILES which are configured when the database software is installed. As I have said, these configurations can be changed by us manually. Because of this back and forth communication, security is very important. This is why we need to monitor and change defaults.

The job of TSNAMES files is to connect target listeners and the database by providing needed services to names and database connection aliases. TSNAMES will also accept a request from the client to set up a database connection. In the default version of the file, as part of installation, we will have "ORACLR\_CONNECTION\_ DATA" service created to provide support to all external services. Here is how that service looks inside:

```
ORACLR_CONNECTION_DATA = (DESCRIPTION =  
(ADDRESS_LIST = (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1521)) )  
(CONNECT_DATA = (SID = CLRExtProc)  
(PRESENTATION = RO) ) )
```

We have two parameters in the example, KEY, and SID, and their values vary unlike the fixed values of other parameters. These values need to live by the values of the matching entries of the listener. The ADDRESS parameter is there to look for listeners that qualify to receive any internet procedure call requests through the EXTPROC1521 value. The PROTOCOL parameter has a fixed value, so it can set up communication between the external service request and the server. The CONNECT\_DATA parameter is used to initiate the extproc process as soon as the ADDRESS is matched to an active listener. The PRESENTATION parameter boosts the performance of the direction

server's response to the client using a remote operations protocol.

Remember that database connections requests go through listeners. This is why we have a listener entry file that contains all the information about the network configurations of the server. The LISTENER file should contain the LISTENER and the SID LIST LISTENER if you are using the default version. LISTENER entry's job is to provide details about the protocol and the key. SID LIST LISTENER contains all the SID information from external services that communicated with the listener. It contains SID NAME, PROGRAM parameters, and ORACLE HOME information. The first has to always be kept in sync with the SID value in order for it to recognize the right extproc process. PROGRAM parameters provide extproxc with program identification capability.

Another important point is that the shared library's file location needs to be registered within the SID LIST LISTENER entry. Oracle has various restrictions on the location of shared libraries. By default, the libraries are located in the \$ORACLE\_HOME\BIN\ directory, so any other locations need to be declared inside the EXTPROC DLLS parameter.

There are only three allowed values in the environment: ANY, ONLY, or the DLL path. The first options allow us to declare DLL files in various locations (paths are stated by separating them with colons). This will provide more security because of the restrictions affecting libraries when being analyzed by extproc. You can specify the actual DLL without using the ONLY parameter. This way, DLL files will be accessible in the default directory. ANY parameter enables all DLLs to be loaded.

Ensure you include a separate listener for the extproc process so that you can isolate the management of the TCP and ICP requests travel through the SID LIST entries. While you're in a secure environment, back up the LISTENERS and TNSNAMES files. It is also advised that you change the LISTENER and SID LIST LISTENER entries for the SID NAME and TCP requests. Lastly, introduce two new entries to the LISTENER file: EXPROC LISTENER and an SID LIST EXTPROC LISTENER. Now let us look at an example which includes all of this:

```
LISTENER = (DESCRIPTION_LIST = (DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)
```

```
(HOST = <<host>>)(PORT = 1521)))  
SID_LIST_LISTENER = (SID_LIST = (SID_DESC = (SID_NAME = <<Database Name>>)  
(ORACLE_HOME = <<Oracle Home>>)))  
EXTPROC_LISTENER = (DESCRIPTION_LIST = (DESCRIPTION = (ADDRESS =  
(PROTOCOL = IPC)(KEY = <<extproc key>>)))  
SID_LIST_EXTPROC_LISTENER = (SID_LIST = (SID_DESC = (SID_NAME =  
CLRExtProc)  
(ORACLE_HOME = <<Oracle Home>>)  
(PROGRAM = EXTPROC1521)  
(ENVS= "EXTPROC_DLLS=[ONLY | ANY | (DLL path)]"))))
```

(Source:

[https://docs.oracle.com/cd/B12037\\_01/network.101/b10776/listener.htm](https://docs.oracle.com/cd/B12037_01/network.101/b10776/listener.htm)  
retrieved in December 2019)

Remember that we need to recreate the original service, so it can inherit all the modifications for a new listener.

External procedures come with many benefits. They put us in a position where we can exploit features of other programming languages to make our tech more flexible, adaptable, and optimized. It also makes client logic reusable because the server-side external applications are shared between users of the database (shared by all of them). The client will also benefit from increased performance because we have a computational task running on the server instead of the client, saving a lot of resources. It is a better experience overall, for everyone involved.

## External Programs

Oracle has expanded its support for external applications written in various programming languages, like C++ or Java. We saw how, through an external procedure, the processing is performed. In this section, we will discuss the development stages we need to go through to execute a program written in C using PL/SQL.

The first step is obvious—you write the C-based program and compile it. You then copy the code file and add it to the default Oracle directory we mentioned in the previous section. We make the DLL using the C compiler and configure the Oracle network service. Now, we need to generate PL/SQL library object from the DLL and create a call specification so that we can publish the external program. Remember the programming language should

be specified, as well as the library name, the external application's method, and all the parameters that need to be mapped. Once this is done we can test the application by running it. Of course, this is a simplified overview, and depending on your project there may be extra steps and other considerations. We will talk about these stages in detail soon.

As you know, when we're in a secure program, the client separates string utility methods from data layers. In our example, we will use a tiny program to transform the case of a string input from lowercase to uppercase.

Let's first illustrate creating and compiling an application. Our program is CaseConvert.c and it will accept two inputs and return their sum.

```
#include<stdio.h>
#include<conio.h>
char ChangeCase(char *caps)
{
    int index = 0;
    while (caps[index])
    {
        caps[index] = toupper(caps[index]);
        index++;
    }
    return caps;
    getch();
}
```

(Source: <https://codescracker.com/c/program/c-program-convert-uppercase-into-lowercase.htm> retrieved in December 2019)

As long as the code file is writable, you can place it in any location on the server. In most real-life scenarios the standard process involves saving the file to the operating system. For the example's sake, we are going to save it at C:\Examples\C\. Remember that the compilation process is very important when it comes to assessing if the program is viable. After compilation, we have CaseConvert.o as a compiled module. It will not be placed in the same location as the initial file. Now, we generate the shared library for the program. We create it with "gcc" like so:

```
C:\Examples\C>gcc -shared CaseConvert.c -o CaseConvert.dll
```

Ensure that the DLL uses the same path as the code file and it is found in the

LISTENER file. Now we are going to configure the Oracle network service and match configurations as we said in the last section. To verify that a KEY value is found under the ADDRESS parameter, case sensitivity has to be followed. Also, the SID value in the CONNECT DATA parameter under the TNS service must match the SID\_NAME in the SID list listener. Finally, the DLL path should be perfectly mapped, so we can create the PL/SQL library object.

The library object will function the same as a database alias that stands for the location of the shared library, which all subprograms will have access to. Any user will be able to create a library as long a privilege is given to them. With that said, let's take a look at the syntax:

```
CREATE [OR REPLACE ] LIBRARY [Library name]
[IS | AS] [DLL path]
AGENT [Agent DB link, if any];
```

The DLL path is going to the DLL's location on the server. Note that an Oracle server will not perform any verification to confirm that the file exists inside the syntax. Below we are creating the library:

```
/*Connect as SYSDBA*/
CONN sys/system AS SYSDBA Connected.
/*Grant the CREATE LIBRARY privilege*/
GRANT CREATE LIBRARY TO ORADEV;
```

(Source: <https://docs.oracle.com/database/121/ADMQS/GUID-DE8A79BD-FAE4-4364-98FF-D2BD992A06E7.htm#ADMQS0361> retrieved in December 2019)

Now, we should publish the external program using a call specification. A wrapper method will be generated to call the external procedure from the database. The library object will be used as a reference for the DLL which holds the application in the form of a linked module. The program method will be found inside; it needs to have the same name as what we used inside the program. The parameter will be mapped based on compatibility between C programming language and PL/SQL. For instance, an integer will be mapped as a NUMBER.

It's worth mentioning that the wrapper method is what we refer to when we

say call specification. We use it when we publish the external applications, but it has several purposes.

1. It allows communications between C and the database engine.
2. It dispatches the application.
3. It manages memory and maintains the database.

Another amazing thing about it is that it can be used as a standalone function, package, or procedure. Its object type depends on the structure of the program. Here's how it looks:

```
CREATE OR REPLACE FUNCTION [Name] [Parameters]
RETURN [data type] [IS | AS] [call specification]
END;
```

(Source:

[https://docs.oracle.com/cd/A64702\\_01/doc/server.805/a58236/10\\_procs.htm](https://docs.oracle.com/cd/A64702_01/doc/server.805/a58236/10_procs.htm)  
retrieved in December 2019)

We can connect the specification to other procedures. Here's how that would look:

```
CREATE OR REPLACE PROCEDURE [Name] [Parameters] [IS | AS]
[Call specification] END;
```

The functions of the call specification are to connect all the libraries' details with the programming language and the external program. This forms a call to a subprogram that can be a function, package, subprogram, or procedure. Here is how a general call specification looks:

```
AS LANGUAGE C [LIBRARY (library name)] [NAME (external program name)]
[WITH CONTEXT] [AGENT IN (formal parameters)]
[PARAMETERS (parameter list)];
```

(Source:

[https://docs.oracle.com/cd/A64702\\_01/doc/server.805/a58236/10\\_procs.htm](https://docs.oracle.com/cd/A64702_01/doc/server.805/a58236/10_procs.htm)  
retrieved in December 2019)

Briefly, let's look at the elements in this code:

1. We have LANGUAGE C which is used to declare the language of the applications. In our example that language happens to be C, but it can be C++ or another language.
2. We have LIBRARY which is our database library object.

3. We have NAME which tells us the name of our external program. Remember that the name has to be case sensitive.
4. We have WITH CONTEXT which is needed to pass context pointers to the invoked program.
5. Last is the number of formal parameters sent to the specifications, and other parameters that represent the mapping between programming language C and PL/SQL.

Now, let's code a call specification:

```
CREATE OR REPLACE FUNCTION
F_CASE_CONVERT (P_STRING VARCHAR2)
/* Declare the RETURN type in compatibility with the external program*/
RETURN VARCHAR2
/*Declare the external program's programming language*/
AS LANGUAGE C
/*Declare the PL/SQL library object name*/
LIBRARY EXTDLL
/*Declare the external function name*/
NAME "ChangeCase"
/*Declare the parameters*/
PARAMETERS (P_STRING STRING);
(Source: https://docs.oracle.com/cd/A64702\_01/doc/server.805/a58236/10\_procs.htm retrieved in December 2019)
Now, we have to test it using the PL/SQL block:
SQL> DECLARE
/*Declare a local parameter. Test data is used to initialize it*/
l_str VARCHAR2(1000) := 'oracle pl/sql developer';
BEGIN
/* Call the function and display the result*/
l_str := F_CASE_CONVERT (l_str);
DBMS_OUTPUT.PUT_LINE(l_str);
END;
```

(Source: [https://docs.oracle.com/cd/A64702\\_01/doc/server.805/a58236/10\\_procs.htm](https://docs.oracle.com/cd/A64702_01/doc/server.805/a58236/10_procs.htm) retrieved in December 2019)

We have now combined Oracle and SQL with a C-based program. Java classes can work as external procedures as well, but they and their source files will be stored in a different format in the database (the schema items). Also, the classes will need to be processed on the logical side instead of the user interface. This shows us that applications that bring simple data

computing are not the best programs to connect to Oracle.

For our sake, code written in Java is easy to call from PL/SQL and minimizes network requirements on the client-side. It also improves logic distribution through all layers, diminishing redundancy in the code.

One of the reasons Java is better at this is because it has no dependency on shared libraries. This does not mean it has no libraries. Java has a sort of a library called Litunit, but unlike C, during the invocation, it gets automatically loaded and run. This makes sense; Java is a native section of the Oracle database. The extproc agent isn't needed to handle all communication between Oracle and Java, because of the Java Virtual Machine which is already a part of Oracle. While Java integration is easier, applications written in C will perform better than those written in Java when PL/SQL is involved.



# CHAPTER 8:

## Large Objects

You have probably seen by now that managing data is not as easy as it seems in theory. You always have to be thinking about several things at the same time – things like security, data integrity, the right storage system, and support for various forms of data. To make things worse, software development is growing at a rapid pace, and with it a need to handle large amounts of binary-based data. Before large objects, LOB, Oracle had to handle large amounts of data with LONG and LONG RAW data.

There are many restrictions one faces when dealing with LONG data types because they require things like stable storage systems. So to avoid headaches, LOBs were created. Large unstructured binary is handled as BLOBS and large character-based large files are CLOBS. There are also BFILE data types that represent the location of dependent binary files. We are going to explore all these large files to gain a better understanding of them. You should be aware that large objects come in many forms; they can be a system-based physical file or a file on the OS like a document or image. With that said, let's begin looking at these data types.

## LOB Data Types

Because working with LONG and LONG raw became cumbersome for database developers, a need arose for a large and stable data type. Here are a few of the problems that were faced by developers:

1. Tables weren't able to contain a single LONG or LONG RAW

column and data had to be deposited inline with the record. This means dumping the same segments frequently.

2. We also could only store 2GB of information in LONG and LONG RAW columns. Considering the amount of data we need to handle today – an amount in terabytes – that is very small.
3. LONG data types only supported sequential data access. Also, other restrictions resulted from association with this data type. For instance, single-column specifications mean we can't rely on indexing which leads to compatibility problems.

So when LOB data types were introduced, these issues were resolved, and LOB data was divided into CLOB, BLOB, BFILE, and NCLOB. All of these data types follow a stable storage system and they can be accessed at random.

Here are more of their features:

1. LOBs can store a maximum of 128 TB worth of data, which is mind-bogglingly large when compared to 2GB.
2. LOBS can be stored in separate sections or different segments of the table.
3. They support national character sets, and they come with the ORACLE database management systems package for LOBs to perform LOB-specific operations.

You can see why when LOBs were introduced they became very popular, while their counterparts became obsolete. But we shouldn't think that old data types are no longer useful—they can still be found inside dictionary views and are often used to deposit free-text data. Oracle has continued to support them even when LOBs are better.

You should know that LOB data, once stored in the database, becomes internal to that database. This means a column of LOB data types is incorporated within the table with a LOB value that is made part of the database. The PL/SQL variable of LOBs is accessed through the PL/SQL block, but CLOBs, BLOBs, and NCLOBs remain internal Oracle data types.

Internal LOBS can be used to declare the type of column or the attribute of a specific object type. Remember that in the PL/SQL block, internal data types are used as local memory variables; in this case, they can be either temporary

or persistent. That cauterization is easy to understand at this point. The LOB information stored inside a table is a persistent LOB, meaning it can be used in a variety of operations involving transactions and selections. On the other hand, temporary internal LOBs are specified and accessed only through the PL/SQL block.

An external LOB is a complex file. An external LOB data type is used to store the locator value instead of storing the actual LOB data. The file can be found on an OS location but it is not a physical part of the database. External LOBS are supported by the BFILE data type. They follow referential semantics to access any database-dwelling external object. This means they cannot be a part of transactions occurring in the database.

It is worth mentioning that LOB structures can be divided into two components: a "what" and a "where." The "what" is the value of the LOB, and the "where" is the locator component. These two elements form the architecture of a LOB.

The value is the larger file that is loaded to the database, and the locators' job is to lead to the location of the file- it is essentially the pointer because it tells us where the file is. The locator is always stored inline with the table regardless of where the value is stored. When LOB data is added, a dedicated column will contain a locator element which will lead to the location of the value. The value is stored in the LOB section and can be deposited in various table spaces. To reiterate, the internal LOB section contains both the locator and value, while external LOBs only contain a locator that points to the external value.

## **BLOB, CLOB, NCLOB, and BFILE**

I've mentioned that there are three types of internal LOB types. Now let us look at them briefly below.

1. BLOB is a binary object data type that is used to store binary files like images, videos, and PDFs. It is similar to LONG RAW data but without the restrictions that plague LONG Raw data as we have already established.
2. CLOBS are LOB data types that are used to store character data in

a character set format. CLOBs are not easily compatible with sets of characters that have an established length. You can think of them as being equivalent to LONG data types, but without the restrictions.

3. NCLOBs are similar to CLOBs but their difference lies in their compatibility. NCLOBs offer support for storing multibyte character set data instead of single-byte data alone, and they can also function with character sets with variable width.

In addition to these data types, we have the BFILE data type. It is used to determine and maintain the reference with an externally located file. The value of the BFILE is a NULL element for each BFILE attribute. However, the locator is different and it contains information that points to the larger file data.

The data file will be externally somewhere – it can be on a solid-state drive, CD, or the operating system itself. BFILES contain read-only data so they cannot be involved in data operations and database transactions. So what will happen if we delete a BFILE? The locator will be dropped and removed from the references, but the file data itself will still exist. Note that the locator always needs to be secure because unauthorized access can lead to big problems. You should secure the location of the file to the server machine and it is recommended you use a timeout system that limits reading the BFILE data. Also, certain processes can be dealt with from the operating system file, such as the management of access permissions, storage space, and more.

PL/SQL variables of CLOBs and BLOBs behave like they are temporary LOBs. These variables are used in designated large object-level processes, so they are stored inside a temporary tablespace. If the temporary object appears in the database in the form of a column value, it is seen as an internal LOB. It's important to be aware because temporary LOBs need to be released as soon as the operations they rely on are finished. When that happens the locator will be invalid.

## **Creating Large Object Types**

Just like with handling other data types, table columns that are reserved for storing large data objects have to be declared as LOB data types. We are going to discuss how LOBs are handled and how to create LOB table columns.

The first important Oracle element is the directory, which we need to access files hosted on the operating system. It gives us an interface version of the file's location path in a form of a directory item. These directories are generated by DBA and they give read and write privileges for every user that has access to them. You should know that directories are considered to be non-schema objects, but they can be used as a form of a security container for various files stored on the client or the server. Here's how to create a directory:

```
CREATE DIRECTORY [DIRECTORY NAME] AS [OS LOCATION PATH]
```

See how the creation process does not involve the validation of a location path on the operating system? This is because a directory can also be generated for a path that doesn't exist. For instance, we can do something like this:

```
/*Connect as SYSDBA*/ SQL> CONN sys/system
AS SYSDBA Connected.
/*Create directory for Labs folder located at the server machine*/
SQL> CREATE DIRECTORY MY_FIRST_DIR AS 'C:\Labs\';
/*Create directory for Labs folder located at the client machine (ORCLClient)*/
SQL> CREATE DIRECTORY MY_CLIENT_DIR AS '\\ORCLClient\Labs\';
Now we can set the privileges for the created directory in the following way:
SQL> GRANT READ, WRITE ON DIRECTORY MY_FIRST_DIR TO ORADEV;
```

(Source:

[https://docs.oracle.com/cd/B19306\\_01/server.102/b14200/statements\\_5007.htm](https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5007.htm)  
retrieved in December 2019)

Now, let's set up a table that will contain a LOB:

```
CREATE TABLE <table name>
(Column list) [LOB (<lobcol1> [, <lobcol2>...])
STORE AS [<lob_segment_name>]
([TABLESPACE <tablespace_name>]
[{ENABLE | DISABLE} STORAGE IN ROW]
[CHUNK <chunk_size>]
[PCTVERSION <version_number>]
```

```
[ { CACHE | NO CACHE [{LOGGING | NOLOGGING}]| CACHE READS [{LOGGING | NOLOGGING}}] [STORAGE {MINEXTENTS | MAXEXTENTS}] [INDEX] [<lob_index_name>] [physical attributes] [<storage_for_LOB_index>] ]]
```

(Source:

[https://docs.oracle.com/cd/B28359\\_01/appdev.111/b28393/adlob\\_tables.htm#](https://docs.oracle.com/cd/B28359_01/appdev.111/b28393/adlob_tables.htm#) retrieved in December 2019)

Now, let us discuss what we did there:

1. We have a LOB data type storage clause which is optional. Then we have the large object segment name which can be declared for the data type column of the table. Know that this section of the table will have the same name as the LOB segment section.
2. Then we have the TABLESPACE which we need for storing the target tablespace for the LOB column.
3. The enable disable STORAGE IN ROW expression guarantees the inline and the out of the line storage of LOB data within the table.
4. Then CHUNK is added to the size of the LOB data type information sections (or chunks). It is followed by caching and logging features.
5. We have STORAGE which is required to provide an extent declaration for the storage clause.
6. Then we have the index which gives the LOB index specification.

The table below will store an employee's ID, a text document, and an image. See the LOB specifications, index, and the clause that declares inline and out of line storage:

```
CREATE TABLE EMP_LOB_DEMO ( EMPID NUMBER, DOC CLOB, IMAGE BLOB ) LOB
(DOC)
--LOB storage clause for DOC STORE AS LOBSEGMENT_DOC_CLOB ( CHUNK 4096 CACHE
STORAGE (MINEXTENTS 2)
INDEX IDX_DOC_CLOB ) LOB (IMAGE)
--LOB storage clause for IMAGE STORE AS LOBSEGMENT_IMG_BLOB ( ENABLE STORAGE
IN ROW CHUNK 4096
CACHE STORAGE (MINEXTENTS 2)
INDEX IDX_IMAGE_BLOB ) /
```

(Source:

<https://resources.oreilly.com/examples/9781849687225/commit/0973cbe59b1expanded=1> retrieved in December 2019)

The metadata of the LOB is put inside the USER LOBS dictionary view. Also, the LOB data section was generated for the LOB data column found in the table.

## Managing Large Objects

In the LOB data type management system we have interaction with the loading interface, various data manipulation methods, and selection types. As I have said, internal LOB data types include CLOB or BLOB data type columns that interact through various interfaces, like the LOB database management system native to Oracle. In PL/SQL, LOB data is managed through the LOB database management system by default because the system offers all kinds of subprograms needed to manipulate and extract LOB data.

There are a few recommendations for managing internal LOB data types:

1. First, the LOB column should be initialized as NULL using the EMPTY BLOB or CLOB functions. Then we can populate the CLOB using a text file through PL/SQL, or directly with SQL. On the other hand, the BLOB is loaded through the PL/SQL.
2. For a majority of operations and processes you should use the database management system LOB module.
3. You can modify LOB data with an UPDATE statement or by using subprograms included in the LOB database management system.

You should know that a BFILE is also a LOB data type but it is not secured, which means it is vulnerable to threats. It is not secure because it is located externally. However, access to it can be controlled from the PL/SQL block by manipulating the directory object.

Remember that you can modify user privileges in the directory location so that a select few can write data and others will have reading privileges. Working with directing objects, we can make the database allow additional security layers for BFILES once the aforementioned security measures are taken. We can secure an operating system file in a way that prevents anyone from accessing it or changing it. This would mean that the BFILE data needs to be populated the same way as internal LOBs, while processes and

operations are handled by the LOB database management system. Know that in a session a user can only open a certain number of files.

It is worth mentioning the session-level static initialization parameter is the one that is charged with dictating how many BFILEs a user can open in one session. By default, a user can open ten files, but the parameter can be changed inside the spfile.ora file. When that number is reached, you will not be allowed to open new files. You use the ALTER SESSION instructions to change the parameter's value:

```
/*Connect as SYSDBA*/  
Conn sys/system as sysdba Connected.  
/*Alter the session to modify the maximum open files in a session*/  
ALTER SESSION SET SESSION_MAX_OPEN_FILES = 25 /
```

(Source:

<https://resources.oreilly.com/examples/9781849687225/commit/0973cbe59b1expanded=1> retrieved in December 2019)

In the example, we have changed that number to 25, so you will be able to open 25 files per session.

As I have mentioned, BFILEs are managed like internal LOB data types. The DBA handles most security at operating system and database levels, while a programmer writes programs that will work with the LOB to populate the locator in the table column. It also handles the file system externally by creating a directory object within the database, which will control access to the directory itself. Remember that the BFILE column will only be initialized when we use the “BFILENAME” function. The function’s role is to set a new reference to the file at the location it exists.

```
FUNCTION BFILENAME(directory IN VARCHAR2, filename IN VARCHAR2) RETURN  
BFILE;
```

In the previous example, the directory had a valid database directory name as well as a file with a valid filename stored in the directory. We used the functions to return the locator, so we can assign it to an appropriate BFILE column. It is done by using the INSERT or UPDATE statement. All external files are accessed in read-only mode when using BFILEs. Although you cannot change them, the locator can be manipulated to point us to another



target.

# CHAPTER 9:

## **Tuning and Compiling**

Regardless of the programming language you use, compilation is one of the powerful predictors of how well your application is going to run. This is why Oracle offers two compilation methods. On top of this improvement, Oracle has introduced new tuning techniques to improve the performance of our databases. These optimizations guarantee that code logic will be boosted. In this chapter, we are going to look at these aspects, and how they affect PL/SQL performance. Other topics will include compiler enhancements, real native compilations techniques, and more.

### **Compilation Methods**

A compiler's job is to translate code written in a high level, abstracted language into code that your machine can understand. Once a database is installed this process is transparent. Until recently, Oracle used an interpretable method of compiling the database application units. When the compiler runs in interpreted mode it will translate PL/SQL program into machine code by storing it inside the database and interpreting it when invoked. A newer version of Oracle has introduced native compilation. It is okay to ask yourself if this is better; we will think about that a bit.

When Oracle introduced the new code compilation method, it also made changes to the native compilation method by borrowing from the C compiler. It means that the native compiler uses the C compiler to translate PL/SQL to C, creating a DLL in the process and storing it in the catalog of the database. Remember that native compilation will support any RAC environment;

previous compilations types didn't do this.

Backing up the database, the generated libraries are stored into the file system designated to operating system utilities. The file system operates with various initialization parameters managed by the DBA. Throughout the years, compilation method has relied on the same techniques too, but the main difference is how the code is being scanned. In interpreted compilation, code scanning happens in the runtime phase. In native compilations, code scanning happens outside the compilation process. This little tweak brings a sizable runtime performance boost, notwithstanding the size of the code and how it is applied.

So, it is not that the performance of a natively compiled program will be better than that of an interpreted compiled program. Many factors determine whether a program will perform better than another one. With that said, the PL/SQL compilation relies on the C compiler to create C code and then put it in a shareable library. The fact that native compilation is widely used doesn't mean the C compiler is not reliable. Many production level databases have not been implemented because of licensing issues, and people just like to avoid pesky admin until they have to. Native compilations make it a lot easier to deal with compilation parameters because they require only a few settings.

Oracle's real native compilation will translate PL/SQL code, put it in shareable libraries, and put the native machine code inside the system space. Just know that when the application is called the first time, machine code will be stored inside the shared memory. This is because from there, it can be called repeatedly without hiccups. Let's look at the advantages of real native compilation:

1. It gets rid of the need for a C compiler.
2. Machine code is inside the systems tablespace. Since shareable libraries are not part of the process there isn't a need for file systems libraries.
3. Configuration is done through one parameter, namely the "PLSQL CODE TYPE".
4. In INTERPRETED mode, PL/SQL code is compiled to its version of machine code. In runtime it is executed by the interpreter. In

native mode, the PL/SQL is changed into machine code during runtime, and it is executed by the database server.

5. Runtime performance is improved. It is faster than the C native compilation process.
6. Real native compilation is used for PL/SQL programs alone because an application written in SQL alone will not offer the best performance. Real native compilation comes with the option of setting either at the system, session, or object level.

Choosing a compilation type will depend on the DBA. You shouldn't wonder much about whether to use interpreted or native compilations as those decisions will lie with the database development administrator. During the development cycle, there will be many stages that have their own unique demands and requirements. The program is often debugged, compiled, tested, and the process is repeated. This is why the compilation process needs to be as fast as possible. As a matter of fact, application units will compile quickly in interpreted mode because compiled code is interpreted during runtime.

After we have compiled our application and it is ready, we need it to come to life. In this case, the application unit execution needs to be fast. Natively compiled units are executed faster than if using interpreted compilation code. This means the native compilations mode is best for after the development stage. So, interpreted code compilations boost the speed of the compilation process, while the native compilation process boosts the app's performance. The good news is that all this can be changed according to your current needs, and according to the development stage you find yourself in.

Compilation methods come with an intercompatibility system where different compiled application units can coexist within the same schema. They can even make calls to each other when needed. These calls will not have a negative effect on performance.

Usually, as I have alluded to, interpreted compilation mode is chosen over the native mode in the development phase because of how great the need to compile and recompile is during multiple tests. In some scenarios, the code has to be debugged at the optimization level, and the interpreted method works well with various optimization methods.

Interpreted compilation is also suitable for applications that need to handle a multitude of SQL statements. This is because the more SQL statements there are, the more time is needed for interpretation. So the main issue with native compilation is the interpretation phase-in performed at runtime affecting performance negatively.

But native compilation is best for after development, like I have said, because the user of our application will need the app to react fast, which it will. Native compilations will be used mainly to handle PL/SQL units because of the procession logic. But once an application contains a huge number of SQL statements, performance will be reduced, even in the native compilation. When dealing with application units that don't need to move between PL/SQL and SQL, the native compilation is the better option.

## **Tuning PL/SQL Code**

Once your database is configured for the best performance, code is the next thing you turn your attention to because it also plays a major part in how well your system will perform. In this section, we are going to focus on ways to improve your code. Keep in mind these improvements and optimizations in the development phase. You shouldn't wait until later to make these changes as that will be more complicated and you might make more mistakes. Here are the things you will need to improve in your code:

- Identifying the correct data type by avoiding casting.
- Implementing modular programming techniques to spread the workload through modularization.
- Use the FORALL function to bind collections and improve operations. Tune conditional statements by placing them in logical sections.
- There are other areas where your code can benefit from tuning other than the ones listed here.

We have established that SQL is an interactive database language and that PL/SQL is its procedural extension. We often debate both when it comes to performance techniques to implement. Keep that in mind when we look at what makes either one of them a strong tool.

SQL is probably the most used language when it comes to databases and dealing with data. SQL's extension offers more flexibility so we can handle a myriad of real-world problems. Remember that SQL statements require an SQL engine to be executed, just as PL/SQL requires its own engine. When we have an SQL statement inside a PL/SQL application, that context is moved to the SQL engines. This switching is what leads to a reduction in performance, so having a large number of SQL statements inside a PL/SQL program makes the application slower and inefficient.

The problem is, we won't be able to perform in certain scenarios without using SQL. This means SQL statements are a must in some circumstances. To avoid a loss in performance, we can do the following things:

1. Not putting SQL statements in a loop or any iterative structure. This means each time the loop runs the switch happens. So avoid it if you can.
2. Use an API to perform all the non-PL/SQL transactions. A subroutine can also be defined to handle transactions

SQL statements inside an executable can also be optimized as single procedures. The PL/SQL runtime engine can perform type conversion of values; for instance, we can assign a numeric value to any string variable, but we can't do it the other way around. We can also have a data value assigned, but we can't assign the string value to it.

Earlier we learned that the Oracle server manages typecasting. A process like this increases the amount of processing, leading to an increase in work time. For this reason, a variable needs to be declared to the correct data type to handle the value we assign to it. Oracle comes with various functions to improve typecasting. We can use "TO NUMBER", "TO CHAR" or "TO DATE" to label a value's conversion. So imagine we have a PL/SQL block, and we have a string variable and a numeric assigned to it inside a loop. In this situation, the server will figure out the type conversion of the assigned numeric value. Let's take a look at some code:

```
/*Set the PLSQL_OPTIMIZE_LEVEL to 1*/  
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1 /  
/*Enable the SERVEROUTPUT to display block results*/  
SET SERVEROUTPUT ON
```

```

/*Start the PL/SQL block*/
DECLARE L_STR VARCHAR2(10);
L_COUNT NUMBER :=0; BEGIN
/*Capture the system time before loop*/
L_COUNT := DBMS_UTILITY.GET_TIME;
/*Start the loop*/
FOR I IN 1..1000000 LOOP
L_STR := 1; END LOOP;
/*Print the time consumed in the operations*/
DBMS_OUTPUT.PUT_LINE('Time
Consumed:'||TO_CHAR(DBMS_UTILITY.GET_TIME - L_COUNT));
END; /

```

(Source: <https://www.oracle.com/technical-resources/articles/database/sql-11g-plsql.html> retrieved in December 2019)

All we need to do is assign a string value to the string variable.

```

/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE L_STR VARCHAR2(10);
L_COUNT NUMBER :=0; BEGIN
/*Capture the system time before loop*/
L_COUNT := DBMS_UTILITY.GET_TIME;
/*Start a loop which assigns fixed string value to a local string variable*/
FOR I IN 1..1000000 LOOP
L_STR := 'A';
END LOOP;
/*Print the time consumed in the operations*/
DBMS_OUTPUT.PUT_LINE('Time
Consumed:'||TO_CHAR(DBMS_UTILITY.GET_TIME - L_COUNT)); END; /

```

(Source: <https://www.oracle.com/technical-resources/articles/database/sql-11g-plsql.html> retrieved in December 2019)

In many cases we will need to declare a number of NOT NULL variables in our applications to protect the program units from various NULL values in the application. The server deals with a lot of work, performing NO NULL tests before assigning every variable. This affects performance in a bad way. It works by assigning a statement result to a variable that is temporary, then the variable will be tested. If a true value is returned, the block will be terminated because of exception error. If it doesn't, the application moves forward. This tells us that a variable must never be declared using NOT

NULL constraints. Let's look at an example:

```
/*Set PLSQL_OPTIMIZE_LEVEL as 1*/
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=1 /
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON
/*Initiate the PL/SQL block*/
DECLARE L_NUM NUMBER NOT NULL := 0;
L_A NUMBER := 10;
L_COUNT NUMBER;
BEGIN /*Capture the start time*/
L_COUNT := DBMS_UTILITY.GET_TIME;
/*Initiate the loop*/
FOR I IN 1..1000000 LOOP
L_NUM := L_A + I;
END LOOP;
/* Process the time difference */
DBMS_OUTPUT.PUT_LINE('Time
Consumed:'||TO_CHAR(DBMS_UTILITY.GET_TIME - L_COUNT)); END; /
Time Consumed: 17
```

(Source:

[https://docs.oracle.com/cd/E18283\\_01/appdev.112/e16760/d\\_output.htm](https://docs.oracle.com/cd/E18283_01/appdev.112/e16760/d_output.htm)  
retrieved in December 2019)

Another alternative is this:

```
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE L_NUM NUMBER;
L_A NUMBER := 10;
L_COUNT NUMBER;
BEGIN
/*Capture the start time*/
L_COUNT := DBMS_UTILITY.GET_TIME;
/*Start the loop*/
FOR I IN 1..1000000
LOOP    L_NUM := L_A + I;
IF L_NUM IS NULL THEN
DBMS_OUTPUT.PUT_LINE('Result cannot be NULL');
EXIT;    END IF; END LOOP;
/*Compute the time difference and display*/
DBMS_OUTPUT.PUT_LINE('Time Consumed:'||TO_CHAR(DBMS_UTILITY.GET_TIME -
L_COUNT));
END; /
Time Consumed:12
```



(Source:

[https://docs.oracle.com/cd/E18283\\_01/appdev.112/e16760/d\\_output.htm](https://docs.oracle.com/cd/E18283_01/appdev.112/e16760/d_output.htm)  
retrieved in December 2019)

As you can see in the second example, less time is spent on the code. We can see, in this example, how much performance we lose by having NOT NULL constraints.

We also have another data type we need to consider, the PLS\_INTEGER. This data type is part of the number category and it was added with Oracle version 7 with the purpose of boosting processing speeds when dealing with complex arithmetic operations. It is the only data type that relies on native machine calculations instead of the C library. This is what allows it to perform so quickly. For instance, a 32-bit sized data type can contain values between -2147483648 and 2147483647. Let's take a look at an example that shows us this difference in performance. We are going to compare the NUMBER type to the PLS\_INTEGER type.

```
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/ DECLARE
L_NUM NUMBER := 0;
L_ST_TIME NUMBER;
L_END_TIME NUMBER; BEGIN
/*Capture the start time*/
L_ST_TIME := DBMS_UTILITY.GET_TIME();
/*Begin the loop to perform a mathematical calculation*/
FOR I IN 1..100000000 LOOP
/*The mathematical operation increments a variable by one*/
L_NUM := L_NUM+1;
END LOOP;
L_END_TIME := DBMS_UTILITY.GET_TIME();
/*Display the time consumed*/
DBMS_OUTPUT.PUT_LINE('Time taken by NUMBER:'||TO_CHAR(L_END_TIME -
L_ST_TIME));
END; /
Time taken by NUMBER:643
(Source: https://docs.oracle.com/cd/A97630\_01/appdev.920/a96624/a\_samps.htm retrieved in
December 2019)
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE L_PLS PLS_INTEGER := 0;
```

```
L_ST_TIME NUMBER;
L_END_TIME NUMBER;
BEGIN
/*Capture the start time*/
L_ST_TIME := DBMS_UTILITY.GET_TIME();
/*Begin the loop to perform a mathematical calculation*/
FOR I IN 1..100000000 LOOP
/*The mathematical operation increments a variable by one*/
L_PLS := L_PLS+1; END LOOP;
/*Display the time consumed*/
L_END_TIME := DBMS_UTILITY.GET_TIME();
DBMS_OUTPUT.PUT_LINE('Time taken by PLS_INTEGER:'||TO_CHAR(L_END_ TIME -
L_ST_TIME));
END; /
Time taken by PLS_INTEGER:196
```

(Source:

[https://docs.oracle.com/cd/A97630\\_01/appdev.920/a96624/a\\_samps.htm](https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/a_samps.htm)  
retrieved in December 2019)

That is a huge difference in time. The NUMBER data type took three times longer to perform the same operation. It is a perfect example of how changing a data type to an optimized type can lead to an increase in performance. So if something feels funny to you, always look for options.

# CONCLUSION

I want you to know that this is only the beginning of the journey. So keep on reading, practicing, building projects, and collaborating with others to grow your skills and knowledge. You will not regret it. Anything that grows you as a person is worth all the effort. We live in a world where data is everything, and we need capable people to store it and manage it properly. You are one of those people.

# REFERENCES

Casteel, J. (2014). Oracle 11g: Sql. Australia: Brooks/Cole.

Duckett, G. (2016). Sql programming: questions and answers. Place of publication not identified: CreateSpace Independent Publishing Platform.

Malik, Goldwasser, M., & Johnston, B. (2019). Sql for data analytics: perform fast and efficient data analysis with the power of Sql. Birmingham etc.: Packt publishing.

Molinaro, A. (2011). Sql cookbook. Beijing: OReilly.