Expert Guides To Master SQL Programming Quickly with Practical Exercises

HENDRIX ALVAREZ

Expert Guides To Master SQL Programming Quickly with Practical Exercises

HENDRIX ALVAREZ

SQL, WHAT SQL?

Overview

Almost every big tech company uses SQL to manage and organize data be it Uber, Netflix, Airbnb, Facebook, Google, LinkedIn, or any other, SQL is everywhere. SQL is also the top and most popular language among data scientists or data engineers.

Despite lots of hype around NoSQL, Hadoop, and other technologies, it's one of the most-used languages in the entire tech industry, and one of the most popular languages for developers of all sorts.

WHAT IS A DATABASE?

Database defined

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data.

WHAT IS STRUCTURED QUERY LANGUAGE (SQL)?

SQL is a programming language used by nearly all relational databases to query, manipulate, and define data, and to provide access control. SQL was first developed at IBM in the 1970s with Oracle as a major contributor, which led to implementation of the SQL ANSI standard, SQL has spurred many extensions from companies such as IBM, Oracle, and Microsoft. Although SQL is still widely used today, new programming languages are beginning to appear.

Evolution of the database

Databases have evolved dramatically since their inception in the early 1960s. Navigational databases such as the hierarchical database (which relied on a tree-like model and allowed only a one-to-many relationship), and the network database (a more flexible model that allowed multiple relationships), were the original systems used to store and manipulate data. Although simple, these early systems were inflexible. In the 1980s, relational databases became popular, followed by object-oriented databases in the 1990s. More recently, NoSQL databases came about as a response to the growth of the internet and the need for faster speed and processing of unstructured data. Today, cloud databases and self-driving databases are breaking new ground when it comes to how data is collected, stored, managed, and utilized.

What's the difference between a database and a spreadsheet?

Databases and spreadsheets (such as Microsoft Excel) are both convenient ways to store information. The primary differences between the two are:

How the data is stored and manipulated
Who can access the data
How much data can be stored

Spreadsheets were originally designed for one user, and their characteristics reflect that. They're great for a single user or small number of users who don't need to do a lot of incredibly complicated data manipulation.

Databases, on the other hand, are designed to hold much larger collections of organized information—massive amounts, sometimes. Databases allow multiple users at the same time to quickly and securely access and query the data using highly complex logic and language.

Types of databases

There are many different types of databases. The best database for a specific organization depends on how the organization intends to use the data.

Relational databases

Relational databases became dominant in the 1980s. Items in a relational database are organized as a set of tables with columns and rows. Relational database technology provides the most efficient and flexible way to access structured information.

Object-oriented databases

Information in an object-oriented database is represented in the form of objects, as in object-oriented programming.

Distributed databases

A distributed database consists of two or more files located in different sites. The database may be stored on

multiple computers, located in the same physical location, or scattered over different networks.

Data warehouses

A central repository for data, a data warehouse is a type of database specifically designed for fast query and analysis.

NoSQL databases

A NoSQL, or nonrelational database, allows unstructured and semistructured data to be stored and manipulated (in contrast to a relational database, which defines how all data inserted into the database must be composed). NoSQL databases grew popular as web applications became more common and more complex.

Graph databases

A graph database stores data in terms of entities and the relationships between entities.

OLTP databases. An OLTP database is a speedy, analytic database designed for large numbers of transactions performed by multiple users.

These are only a few of the several dozen types of databases in use today. Other, less common databases are tailored to very specific scientific, financial, or other

functions. In addition to the different database types, changes in technology development approaches and dramatic advances such as the cloud and automation are propelling databases in entirely new directions. Some of the latest databases include

USING QUERIES TO OBTAIN DATA

Structured Query Language (SQL) is a specialized language for updating, deleting, and requesting information from databases. SQL is an ANSI and ISO standard, and is the de facto standard database query language. A variety of established database products support SQL, including products from Oracle and Microsoft SQL Server. It is widely used in both industry and academia, often for enormous, complex databases.

In a distributed database system, a program often referred to as the database's "back end" runs constantly on a server, interpreting data files on the server as a standard relational database. Programs on client computers allow users to manipulate that data, using tables, columns, rows, and fields. To do this, client programs send SQL statements to the server. The server then processes these statements and returns result sets to the client program.

THE DATA DEFINITION LANGUAGE (DDL)

Data definition language (DDL) is a computer language used to create and modify the structure of database objects in a database. These database objects include views, schemas, tables, indexes, etc.

This term is also known as data description language in some contexts, as it describes the fields and records in a database tab.

SQL JOINS AND UNION

A data definition language (DDL) is a computer language used to create and modify the structure of database objects in a database. These database objects include views, schemas, tables, indexes, etc.

This term is also known as data description language in some contexts, as it describes the fields and records in a database table.

DATA INTEGRITY

Data integrity is the maintenance of, and the assurance of, data accuracy and consistency over its entire life-cycle and is a critical aspect to the design, implementation, and usage of any system that stores, processes, or retrieves data. The overall intent of any data integrity technique is the same: ensure data is recorded exactly as intended (such as a database

correctly rejecting mutually exclusive possibilities). Moreover, upon later retrieval, ensure the data is the same as when it was originally recorded. In short, data integrity aims to prevent unintentional changes to information. Data integrity is not to be confused with data security, the discipline of protecting data from unauthorized parties.

Any unintended changes to data as the result of a storage, retrieval or processing operation, including malicious intent, unexpected hardware failure, and human error, is failure of data integrity. If the changes are the result of unauthorized access, it may also be a failure of data security.

CREATING AN SQL VIEW

Most of the time, views can be defined as “virtual or logical” tables, but if we expand this basic definition we can understand the views more clearly. A view is a query that is stored in the database and returns the result set of the query in which it is defined. The query that defines the view can be composed of one or more tables. A view returns column or columns of the query in which it is referenced. However, we need to underline a significant point about the views, a simple view never stores data, merely it fetches the results of the query in which it is defined.

TABLE OF CONTENTS

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 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.

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

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 90th 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 planed 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:

officebranch	paintball	archery	football
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:

cust_id	first_nm	second_nm	lastname	sex	Date_of_birth
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:

cust_id	first_nm	second_nm	lastname	sex	Date_of_birth
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	Coventry
03231	Juan		Maslow	Female	1978-11-18	York

03146	John	Ken	Pascal	Male	1983-07-12	Liverpool
03347	Alan		Basal	Male	1975-10-09	Easton

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-2[2]
```

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:

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 4.6 London_Customers

acct_no	first_name	Surname	sex	acct_bal
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_Customers[2]
```

The resulting table will look like this:

first_name	surname	sex	acct_bal
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:

first_name	surname	sex	acct_bal
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 the 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_name[2]
```

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_statemnt
 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_name [2]
```

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 datatyle[NULL| NOT NULL].
 column1 datatyle[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 Meadows, California

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.

First Name	Last Name	Position	Salary	Address
James	Hunt		7,500.00	

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

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 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	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 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 View, New York	NULL
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 factional 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` constrains 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.

	BackupName	BackupDescription	BackupType	ExpirationDate	Compressed	Position
1	NULL	NULL	1	NULL	0	1
2	NULL	NULL	1	NULL	0	2
3	NULL	NULL	1	NULL	0	3

4	NULL	NULL	1	NULL	0	4
5	NULL	NULL	1	NULL	0	5
6	NULL	NULL	1	NULL	0	6
7	NULL	NULL	5	NULL	0	7
8	NULL	NULL	1	NULL	0	8
9	NULL	NULL	5	NULL	0	9

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:

ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size
317	LL Crankarm	CA-5965	Black	0	0	NULL

318	ML Crankarm	CA-6738	Blac k	0	0	NULL
19	HL Crankarm	CA-7457	Blac k	0	0	NULL
320	Chainring Bolts	CB-2903	Silve r	0	0	NULL
321	Chainring Nuts	CN-6137	Silve r	0	0	NULL

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]
GO
```

Assigning Server Roles

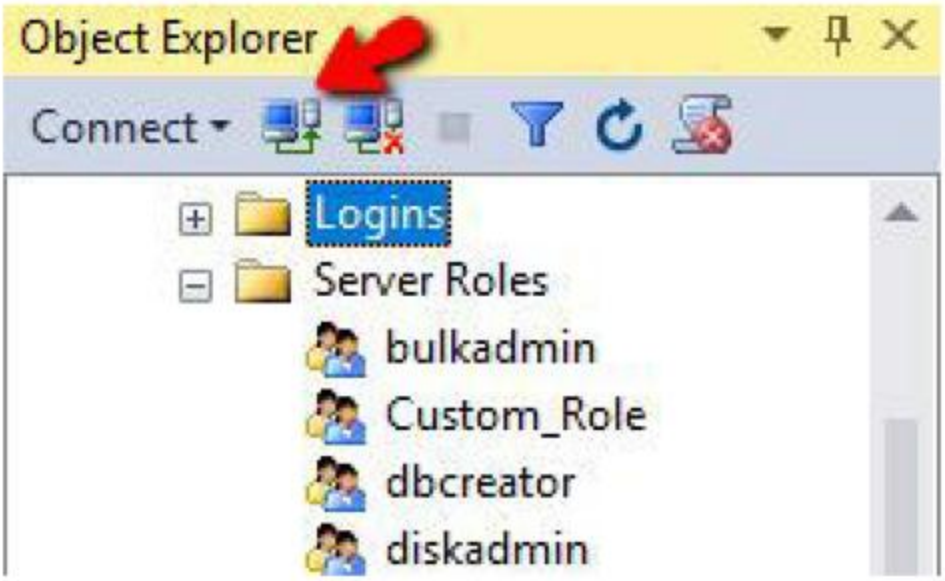
In order to assign the server level roles, we can use several methods. One of the more popular ones involves using the *Management Studio* interface, and another one is using pure syntax. Let's first start with the syntax. In the following example we are going to give the user the role of `Dbcreator`. 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.

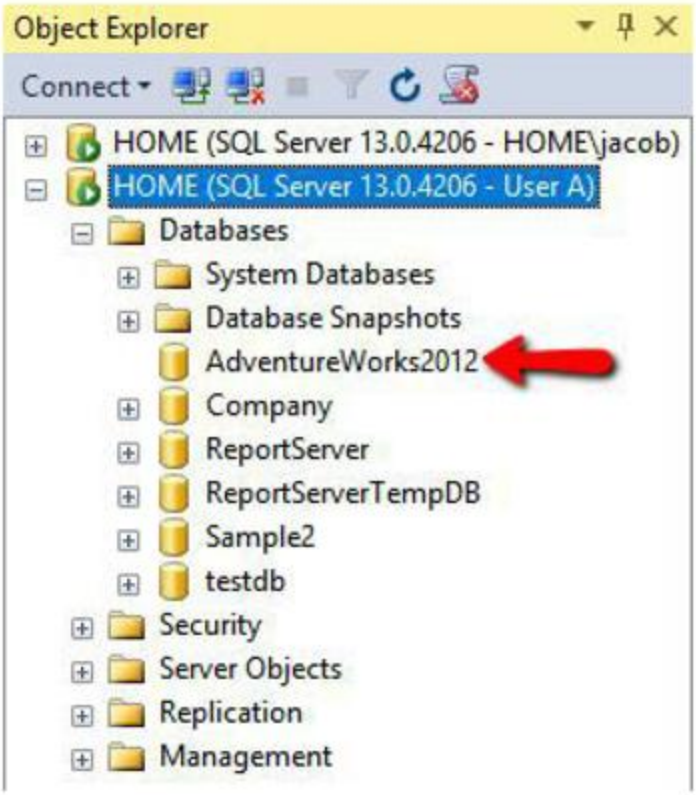
Now, let's look at the *Management Studio* option. In order to log in, head to the Connect Object Explorer button and click on it to see the following connection window.



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.

Image 124


Image 125


Image 126


Image 127


Image 128


Image 129


Image 130


Image 131


Image 132


Image 133


Image 134


Image 135


Image 136


Image 137


Image 138


Image 139


Image 140


Image 141


Image 142


Image 143


Image 144


Image 145


Image 146


Image 147


Image 148


Image 149


Image 150


Image 151


Image 152


Image 153


Image 154


Image 155


Image 156


Image 157


Image 158


Image 159


Image 160


Image 161


Image 162


Image 163


Image 164


Image 165


Image 166


Image 167


Image 168


Image 169


Image 170


Image 171


Image 172


Image 173


Image 174


Image 175


Image 176


Image 177


Image 178


Image 179


Image 180


Image 181


Image 182


Image 183


Image 184


Image 185


Image 186


Image 187


Image 188


Image 189


Image 190


Image 191


Image 192


Image 193


Image 194


Image 195


Image 196


Image 197


Image 198


Image 199


Image 200


Image 201


Image 202


Image 203


Image 204


Image 205


Image 206


Image 207


Image 208


Image 209


Image 210


Image 211


Image 212


Image 213


Image 214


Image 215


Image 216


Image 217


Image 218


Image 219


Image 220


Image 221


Image 222


Image 223


Image 224


Image 225


Image 226


Image 227


Image 228


Image 229


Image 230


Image 231


Image 232


Image 233


Image 234


Image 235


Image 236


Image 237


Image 238


Image 239


Image 240


Image 241


Image 242


Image 243


Image 244


Image 245


Image 246


Image 247


Image 248


Image 249


Image 250


Image 251


Image 252


Image 253


Image 254


Image 255


Image 256


Image 257


Image 258


Image 259


Image 260


Image 261


Image 262


Image 263


Image 264


Image 265


Image 266


Image 267


Image 268


Image 269


Image 270


Image 271


Image 272


Image 273


Image 274


Image 275


Image 276


Image 277


Image 278


Image 279


Image 280


Image 281


Image 282


Image 283


Image 284


Image 285


Image 286


Image 287


Image 288


Image 289


Image 290


Image 291


Image 292


Image 293


Image 294


Image 295


Image 296


Image 297


Image 298


Image 299


Image 300


Image 301


Image 302


Image 303


Image 304


Image 305


Image 306


Image 307


Image 308


Image 309


Image 310


Image 311


Image 312


Image 313


Image 314


Image 315


Image 316


Image 317


Image 318


Image 319


Image 320


Image 321


Image 322


Image 323


Image 324


Image 325


Image 326


Image 327


Image 328


Image 329


Image 330


Image 331


Image 332


Image 333


Image 334


Image 335


Image 336


Image 337


Image 338


Image 339


Image 340


Image 341


Image 342


Image 343


Image 344


Image 345


Image 346


Image 347


Image 348


Image 349


Image 350


Image 351


Image 352


Image 353


Image 354


Image 355


Image 356


Image 357


Image 358


Image 359


Image 360


Image 361


Image 362


Image 363


Image 364


Image 365


Image 366


Image 367


Image 368


Image 369


Image 370


Image 371


Image 372


Image 373


Image 374


Image 375


Image 376


Image 377


Image 378


Image 379


Image 380


Image 381


Image 382


Image 383


Image 384


Image 385


Image 386


Image 387


Image 388


Image 389


Image 390


Image 391


Image 392


Image 393


Image 394


Image 395


Image 396


Image 397


Image 398


Image 399


Image 400


Image 401


Image 402


Image 403


Image 404


Image 405


Image 406

DOCUMENT OUTLINE

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_012
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_013
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_014
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_015
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_016
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_017
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_018
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_019
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_020
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_021
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_022
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_023
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_024

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_025
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_026
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_027
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_028
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_029
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_030
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_031
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_032
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_033
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_034
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_035
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_036
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_037

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_038
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_039
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_040
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_041
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_042
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_043
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_044
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_045
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_046
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_047
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_048
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_049
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_050

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_051
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_052
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_053
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_054
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_055
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_056
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_057
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_058
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_059
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_060
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_061
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_062
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_063

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_064
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_065
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_066
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_067
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_068
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_069
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_070
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_071
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_072
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_073
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_074
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_075
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_076

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_077
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_078
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_079
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_080
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_081
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_082
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_083
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_084
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_085
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_086
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_087
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_088
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_089

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_090
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_091
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_092
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_093
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_094
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_095
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_096
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_097
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_098
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_099
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_100
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_101
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_102

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_103
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_104
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_105
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_106
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_107
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_108
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_109
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_110
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_111
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_112
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_113
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_114
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_115

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_116
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_117
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_118
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_119
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_120
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_121
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_122
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_123
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_124
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_125
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_126
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_127
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_128

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_129
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_130
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_131
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_132
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_133
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_134
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_135
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_136
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_137
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_138
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_139
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_140
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_141

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_142
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_143
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_144
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_145
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_146
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_147
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_148
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_149
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_150
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_151
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_152
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_153
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_154

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_155
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_156
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_157
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_158
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_159
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_160
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_161
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_162
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_163
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_164
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_165
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_166
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_167

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_168
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_169
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_170
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_171
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_172
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_173
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_174
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_175
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_176
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_177
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_178
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_179
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_180

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_181
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_182
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_183
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_184
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_185
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_186
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_187
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_188
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_189
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_190
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_191
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_192
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_193

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_194
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_195
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_196
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_197
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_198
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_199
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_200
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_201
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_202
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_203
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_204
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_205
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_206

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_207
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_208
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_209
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_210
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_211
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_212
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_213
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_214
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_215
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_216
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_217
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_218
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_219

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_220
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_221
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_222
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_223
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_224
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_225
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_226
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_227
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_228
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_229
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_230
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_231
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_232

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_233
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_234
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_235
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_236
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_237
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_238
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_239
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_240
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_241
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_242
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_243
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_244
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_245

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_246
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_247
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_248
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_249
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_250
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_251
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_252
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_253
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_254
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_255
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_256
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_257
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_258

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_259
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_260
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_261
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_262
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_263
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_264
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_265
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_266
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_267
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_268
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_269
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_270
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_271

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_272
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_273
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_274
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_275
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_276
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_277
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_278
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_279
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_280
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_281
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_282
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_283
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_284

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_285
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_286
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_287
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_288
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_289
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_290
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_291
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_292
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_293
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_294
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_295
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_296
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_297

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_298
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_299
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_300
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_301
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_302
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_303
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_304
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_305
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_306
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_307
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_308
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_309
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_310

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_311
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_312
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_313
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_314
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_315
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_316
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_317
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_318
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_319
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_320
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_321
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_322
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_323

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_324
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_325
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_326
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_327
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_328
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_329
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_330
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_331
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_332
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_333
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_334
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_335
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_336

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_337
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_338
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_339
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_340
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_341
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_342
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_343
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_344
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_345
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_346
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_347
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_348
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_349

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_350
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_351
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_352
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_353
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_354
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_355
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_356
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_357
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_358
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_359
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_360
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_361
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_362

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_363
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_364
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_365
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_366
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_367
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_368
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_369
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_370
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_371
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_372
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_373
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_374
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_375

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_376
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_377
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_378
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_379
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_380
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_381
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_382
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_383
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_384
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_385
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_386
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_387
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_388

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_389
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_390
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_391
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_392
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_393
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_394
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_395
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_396
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_397
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_398
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_399
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_400
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_401

- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_402
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_403
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_404
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_405
- 1007_1_Expert Guides To Master SQL Programming_Kindle_Page_406