SQL

PROGRAMMING

A Comprehensive Beginner's Tutorial for Learning SQL Step by Step

CHLOE ANNABLE

SQL Programming

A Comprehensive Beginner's Tutorial for Learning SQL Step by Step

TABLE OF CONTENTS

contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

INTRODUCTION

"Never write when you can talk. Never talk when you can nod. And never put anything in an e-mail." – Eliot Spitzer

On a hard disk, data can be stored in different file formats. It can be stored in the form of text files, word files, mp4 files, etc. However, a uniform interface that can provide access to different types of data under one umbrella in a robust and efficient manner is required. Here, the role of databases emerges.

The definition of a database is "a collection of information stored in computer in a way that it can easily be accessed, managed and manipulated."

Databases store data in the form of a collection of tables where each table stores data about a particular entity. The information that we want to store about students will be represented in the columns of the table. Each row of the table will contain the record of a particular student. Each record will be distinguished by a particular column, which will contain a unique value for each row.

Suppose you want to store the ID, name, age, gender, and department of a student. The table in the database that will contain data for this student will look like this:

SID	SName	SAge		SGender	SDepartment
1	Tom	14		Male	Computer
2	Mike	12		Male	Electrical
3	Sandy	13		Female	Electrical
4	Jack	10		Male	Computer
5	Sara	11		Female	Computer

Student Table

Here, the letter “S” has been prefixed with the name of each column. This is just one of the conventions used to denote column names. You can give any name to the columns. (We will look at how to create tables and columns within it in the coming chapters.) It is much easier to access, manipulate, and manage data stored in this form. SQL queries can be executed on the data stored in the form of tables that have relationships with other tables.

A database doesn’t contain a single table. Rather, it contains multiple related tables. Relationships maintain database integrity and prevent data redundancy. For instance, if the school decides to rename the Computer department from “Computer” to “Comp & Soft,” you will have to update the records of all students in the Computer department. You will have to update the 1st, 4th, and 5th records of the student table.

It is easy to update three records; however, in real life scenarios, there are thousands of students and it is an uphill task to update the records of all of them. In such scenarios, relationships between data tables become important. For instance, to solve the aforementioned redundancy problem, we can create another table named Department and store the records of all the departments in that table. The table will look like this:

DID	DName	DCapacity
101	Electrical	800
102	Computer	500
103	Mechanical	500

Department Table

Now, in the student table, instead of storing the department name, the department ID will be stored. The student table will be updated like this:

SID	SName	SAge	SGender	DID
1	Tom	14	Male	102
2	Mike	12	Male	101
3	Sandy	13	Female	101
4	Jack	10	Male	102
5	Sara	11	Female	102

Table Student

You can see that the department name column has been replaced by the department id column, represented by “DID”. The 1st, 4th, and 5th rows that were previously assigned the “Computer” department now contain the id of the department, which is 102. Now, if the name of the department is changed from “Computer” to “Comp & Soft”, this change has to be made only in one record of the department table and all the associated students will be automatically referred to the updated department name.

Advantages of Databases

The following are some of the major advantages of databases:

- Databases maintain data integrity. This means that data changes are carried out at a single place and all the entities accessing the data get the latest version of the data.
- Through complex queries, databases can be efficiently accessed, modified, and manipulated. SQL is designed for this purpose.

- Databases avoid data redundancy. Through tables and relationships, databases avoid data redundancy and data belonging to particular entities resides in a single place in a database.
- Databases offer better and more controlled security. For example, usernames and passwords can be stored in tables with excessive security levels.

Types of SQL Queries

On the basis of functionality, SQL queries can be broadly classified into a couple of major categories as follows:

- **Data Definition Language (DDL)**

Data Definition Language (DDL) queries are used to create and define schemas of databases. The following are some of the queries that fall in this category:

1. CREATE – to create tables and other objects in a database
2. ALTER – to alter database structures, mainly tables.
3. DROP - delete objects, mainly tables from the database

- **Data Manipulation Language**

Data Manipulation Language (DML) queries are used to manipulate data within databases. The following are some examples of DML queries.

1. SELECT – select data from tables of a database
2. UPDATE - updates the existing data within a table
3. DELETE - deletes all rows from a table, but the space for the record remains

CHAPTER 1: The Data Definition Language (DDL)

"Every man has a right to his opinion, but no man has a right to be wrong in his facts. "– Bernard Mannes Baruch

SQL data definition language is used to define new databases, data tables, delete databases, delete data tables, and alter data table structures with the following key words; create, alter and drop. In this chapter, we will have a detailed discussion about the SQL Data Definition Language in a practical style.

DDL for Database and Table Creation

The database creation language is used to create databases in a database management system. The language syntax is as described below:

CREATE DATABASE *my_Database*

For example, to create a customer_details database in your database management system, use the following SQL DDL statement:

CREATE DATABASE customer_details

Please remember that SQL statement is case insensitive. Next, we need to create the various customer tables that will hold the related customers records, in the earlier created 'customer_details' database. This is why the system is called a relational database management system, as all tables are related for easy record retrieval and information processing. To create the different but related customer tables in the customer_details database, we apply the following DDL syntax:

```
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
)
```
[1]

The attribute “PRIMARY KEY” ensures the column named ‘acct_no’ has unique values throughout, with no null values. Every table should have a primary key column to uniquely identify each record of the table. Other column attributes are ‘NOT NULL’ which ensures that a null value is not accepted into the column, and ‘FOREIGN KEY’ which ensures that a related record in another table is not mistakenly or unintentionally deleted. A column with a ‘FOREIGN KEY’ attribute is a copy of a primary key column in another related table. For example, we can create another table ‘customer_personal_info’ in our ‘customer_details’ database like below:

```
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)

)[2]

The newly created ‘customer_personal_info’ table has a primary key column named ‘cust_id’. The ‘customer_accounts’ table needs to include a column named ‘cust_id’ in its field to link to the ‘customer_personal_info’ table in order to access the table for more information about the customer with a given account number.

Therefore, the best way to ensure data integrity between the two tables, so that an active record in a table is never deleted is to insert a key named ‘cust_id’ in the ‘customer_accounts’ table as a foreign key. This ensures that a related record in ‘customer_personal_info’ to another in ‘customer_accounts’ table is never accidentally deleted. We will discuss how to go about this in the next section.

Alter DDL for Foreign Key Addition

Since ‘customer_accounts’ table is already created, we need to alter the table to accommodate the new foreign key. To achieve this, we use the SQL Data Definition Language syntax described below:

ALTER TABLE *mytable*

ADD FOREIGN KEY (targeted_column)

REFERENCES related_table(related_column)

Now, to add the foreign key to the the ‘customer_accounts’ table and make it reference the key column ‘cust_id’ of table ‘customer_personal_info’, we use the following SQL statements:

ALTER TABLE customer_accounts

ADD FOREIGN KEY (cust_id)

REFERENCES customer_personal_info(cust_id)

Foreign Key DDL in Tables

In situations where we need to create foreign keys as we create new tables, we make use of the following DDL syntax:

CREATE TABLE my_table

(

Column-1 data_type FOREIGN KEY, REFERENCES (related column)

)

Unique Constraint DDL in Tables

The unique constraint can be placed on a table column to ensure that the values stored in the column are unique, just like the primary key column. The only difference is that you can only have one primary key column in a table but as many unique columns as you like. The DDL syntax for the creation of a unique table column or field is as described below:

CREATE TABLE my_table

(

Column-1 data_type UNIQUE

)

Delete and Drop DDL in Tables

DROP TABLE *my_table*

It must be noted that this action is irreversible and plenty of care must be taken before doing this. Also, a database can be deleted with the following DDL syntax:

DROP DATABASE *my_database*

This action is also irreversible. At the atomic level, you may decide to delete a column from a data table. To achieve this, we use the 'Delete' DDL rather than 'DROP'. The syntax is as below:

DELETE column_name *FROM* data_table

DDL to Create Views

CREATE VIEW virtual_table AS

SELECT column-1, column-2, …, column-n

FROM data_table

WHERE column-n operator value[3]

For example, given a table (customer_personal_info) like the one below:

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

To create a view of female customers from the table, we use the following SQL Create View statement:

CREATE VIEW [Female Depositors] AS

SELECT cust_id, first_nm, second_nm, lastname, sex, date_of_birth

FROM customer_personal_info

WHERE sex = 'Female'[4]

To query and display the records of the created view, use the following select statement:

*SELECT * FROM* [Female Depositors]

The execution of the above select statement would generate the following view:

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

However, it must be noted that a view is never stored in memory but recreated when needed. A view can be processed just like you would process a real table.

CHAPTER 2: SQL Joins and Union

"I have noted that persons with bad judgment are most insistent that we do what they think best". – Lionel Abe

The select statement can be made to process more than one table at the same time in a single select statement with the use of the logical operators ('OR' and 'AND'). It can sometimes be more efficient to use the Left, right and the inner join operator for more efficient processing.

1. SQL INNER JOIN

The SQL 'INNER JOIN' can also be used to process more than one data table in an SQL query or statement. However, the data tables must have relating columns (the primary and its associated foreign key columns). The "INNER JOIN" keyword returns records from two data tables if a match is found between columns in the affected tables. The syntax of usage is as described below:

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

For example, to select customer acct_no, first_name, surname, sex and account_bal data from across the two tables 'customer_accounts' (table 4.1) and 'customer_personal_info' (table 4.2) based on matching columns 'cust_id', we use the following SQL INNER JOIN query:

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[5]

The above SQL query would produce the result set in table 4.3 below:

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

SQL RIGHT JOIN

When used with the SQL select statement, The “RIGHT JOIN” keyword includes all records in the right data table even when no matching records are found in the left data table. The syntax of usage is as described below:

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 example, to select customers' acct_no, first_name, surname, sex and account_bal details across the two tables, customer_accounts and customer_personal_info display customer personal information whether they have an active account or not. We use the following SQL 'RIGHT JOIN' query:

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_id[6]

The output of execution of the above SQL RIGHT JOIN query is presented in table 4.4 below:

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

SQL LEFT JOIN

The 'LEFT JOIN' is used with the SQL select statement to return all records in the left data table (first table) even if there were no matches found in the relating right table (second table). The syntax of usage is as described below:

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

For example, to display all active accounts holders' details across the 'customer_accounts and 'customer_personal_info' tables, we use the following SQL 'LEFT JOIN' query:

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[7]

The above SQL LEFT JOIN query would produce the result set in table 4.5 below:

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

SQL UNION Command

Two or more SQL select statements' result sets can be joined with the 'UNION' command. The syntax of usage is as described below:

SELECT column-1, column-2… column-n

FROM data_table-1

UNION

SELECT column-1, column-2… column-n

FROM data_table-2

Source[8]

For example, to display a list of our phantom bank's customers in UK and US branches, you may display one record for customers that have accounts in both branches according to the following tables 4.6 and 4.7:

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

We use the following SQL query:

SELECT first_name, surname, sex, acct_bal

FROM London_Customers

UNION

SELECT first_name, surname, sex, acct_bal

FROM Washington_Customers[9]

The following is the result set from the execution of the above SQL query:

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

Note that the record for a customer (John Pascal) is only listed once, even if he is a customer of the two branches. This is because the UNION command lists only distinct records across tables. So, to display all records across associated tables, use 'UNION ALL' instead.

SQL UNION ALL Command

The UNION ALL command is basically the same as the UNION command, except that its displays all records across unionized tables, as explained earlier. The syntax of usage is as described below:

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

Apply the 'UNION ALL' command on the data of the two tables 'London_Customers' and 'Washington_Customers' with the SQL query below:

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

SELECT first_name, surname, sex, acct_bal

FROM Washington_Customers[10]

The records would then be displayed in two tables, as shown below:

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 3: How to Ensure Data Integrity

“The possession of facts is knowledge, the use of them is wisdom.” – Thomas Jefferson

SQL databases don't just store information. If the information's integrity has been compromised, its reliability becomes questionable. If the data is unreliable, the database that contains it also becomes unreliable.

To secure data integrity, SQL offers a wide range of rules that can limit the values a table can hold. These rules, known as "integrity constraints," work on columns and tables. This chapter will explain each kind of constraint.

Integrity Constraints – The Basics

SQL users divide integrity constraints into the following categories:

The Assertions – You need to define this constraint inside a separate definition (which is called the “assertion definition”). This means that you don't indicate an assertion in your table's definition. In SQL, you may apply an assertion to multiple tables.

The Table-Related Constraints – This is a constraint that you need to define inside a table's definition. You may define a constraint as a component of a table or column's definition.

The Domain Constraints – Similar to the assertions, you need to create domain constraints in a separate definition. This kind of constraint works on the column/s that you declared inside the domain involved.

Table-related constraints offer various constraint options. Consequently, these days, it is the most popular category of integrity constraints. You can divide this category into two: column constraints and table constraints. The former belong to the definition

of a column. The latter, on the other hand, act as elements of a table.

The table and column constraints work with different kinds of constraints. The domain constraints and assertions, however, can only work with one constraint type.

The Not Null Constraint

In the previous chapter, you learned that “null” represents an unknown/undefined value. Keep in mind that undefined/unknown is different from zeroes, blanks, default values, and empty strings. Rather, it signifies the absence of a value. You may consider this value as a “flag” (i.e. a bit, number, or character that expresses some data regarding a column). If you leave a column empty, and the value is therefore null, the system will place the “flag” to indicate that it’s an unknown value.

Columns have an attribute called “nullability.” This attribute shows whether the columns can take unknown values or not. In SQL, columns are set to take null values. However, you may change this attribute according to your needs. To disable the nullability of a column, you just have to use the NOT NULL constraint. This constraint informs SQL that the column won't accept any null value.

In this language, you need to use NOT NULL on a column. That means you can't use this constraint on an assertion, domain constraint, or table-based constraint. Using NOT NULL is a simple process. Just add the syntax below to your column definition:

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

As an example, let's assume that you need to generate a table called FICTION_NOVEL_AUTHORS. This table needs to have three columns: AUTHOR_ID, AUTHOR_NAME, and AUTHOR_DOB.

You need to ensure that each entry you add has values for AUTHOR_ID and AUTHOR_NAME. To accomplish this, you must insert the NOT NULL constraint into the definition of both columns. Here's the code:

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

[11]

As you can see, this code did not set NOT NULL for the AUTHOR_DOB column. Consequently, if a new entry doesn't have any value for AUTHOR_DOB, the system will insert a null value in that column.

The Unique Constraint

Table and column constraints accept unique constraints. In SQL, unique constraints belong to one of these two types:

1. UNIQUE
2. PRIMARY KEY

Important Note: This part of the book will concentrate on the first type. You'll learn about the second one later.

Basically, you can use UNIQUE to make sure that a column does not accept duplicate values. This constraint will stop you from entering a value that already exists in the column.

Let's assume that you want to apply this constraint on the AUTHOR_DOB column. This way, you can make sure that the values inside that column are all unique. Now, let's say you realized that requiring dates of birth to be unique is a bad idea since people may be born on the same date. You may adjust your approach by placing the UNIQUE constraint on AUTHOR_NAME

and AUTHOR_DOB. Here, the table will stop you from repeating an AUTHOR_NAME/AUTHOR_DOB pair. You may repeat values in the AUTHOR_NAME and AUTHOR_DOB columns. However, you can't reenter an exact pair that already exists in the table.

Keep in mind that you may tag UNIQUE constraints as table constraints or column constraints. To generate column constraints, add them to the definition of a column. Here is the syntax:

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

If you need to use the UNIQUE constraint on a table, you must insert it into the table definition as an element. Here is the code:

{ CONSTRAINT (name of constraint) }

UNIQUE < (name of column) { [, (name of column)] … } >

As the syntax above shows, using UNIQUE on a table is more complicated than using the constraint on a column. However, you cannot apply UNIQUE on multiple columns. Regardless of how you use this constraint (i.e. either as a table constraint or a column constraint), you may define any number of UNIQUE constraints within each table definition.

Let's apply this constraint on a columnar level:

CREATE TABLE BOOK_LIBRARY

(AUTHOR_NAME CHARACTER (50) ,

BOOK_TITLE CHARACTER (70) UNIQUE,

PUBLISHED_DATE INT) ;[12]

You may also use UNIQUE on other columns. However, its result would be different than if we had used a table constraint on multiple columns. The following code will illustrate this idea:

CREATE TABLE BOOK_LIBRARY

```
( AUTHOR_NAME CHARACTER (50) ,
BOOK_TITLE CHARACTER (70) ,
PUBLISHED_DATE INT,
CONSTRAINT UN_AUTHOR_BOOK UNIQUE ( AUTHOR_NAME, BOOK_TITLE ) ) ;
```
[13]

Now, for the table to accept a new entry, the AUTHOR_NAME and BOOK_TITLE columns must have unique values.

As you've read earlier, the UNIQUE constraint ensures that one or more columns do not have duplicate values. That is an important rule to remember. However, you should also know that UNIQUE doesn't work on “null.” Thus, a column will accept any number of null values even if you have set a UNIQUE constraint on it.

If you want to set your columns not to accept a null value, you must use NOT NULL. Let's apply NOT NULL on the column definition of BOOK_TITLE:

```
CREATE TABLE BOOK_LIBRARY
( AUTHOR_NAME CHARACTER (50) ,
BOOK_TITLE CHARACTER (70) UNIQUE NOT NULL,
PUBLISHED_DATE INT ) ;
```
[14]

In SQL, you may also insert NOT NULL into column definitions that a table-level constraint is pointing to:

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

Source https://www.w3schools.com/sql/sql_notnull.asp

In both cases, the BOOK_TITLE column gets the constraint. That means BOOK_TITLE won't accept null or duplicate values.

The PRIMARY KEY Constraint

The PRIMARY KEY constraint is almost identical to the UNIQUE constraint. You may use a PRIMARY KEY to prevent duplicate entries. In addition, you may apply it to multiple columns and use it as a table constraint or a column constraint. The only difference is that PRIMARY KEY has two distinct restrictions. These restrictions are:

If you apply PRIMARY key on a column, that column won't accept any null value. Basically, you won't have to use the NOT NULL constraint on a column that has PRIMARY KEY.

A table can't have multiple PRIMARY KEY constraints.

These restrictions exist because primary keys (also known as “unique identifiers”) play an important role in each table. As discussed in the first chapter, tables cannot have duplicate rows. This rule is crucial since the SQL language cannot identify redundant rows. If you change a row, all of its duplicates will also be affected.

You need to choose a primary key from the candidate keys of your database. Basically, candidate keys are groups of columns that identify rows in a unique manner. You may enforce a candidate key's uniqueness using UNIQUE or PRIMARY KEY. However, you must place one primary key on each table even if you did not define any unique constraint. This requirement ensures the uniqueness of each data row.

To define a primary key, you need to indicate the column/s you want to use. You can complete this task through PRIMARY KEY (i.e. the SQL keyword). This process is similar to the one

discussed in the previous section. To apply PRIMARY KEY on a new column, use the following syntax:

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

To use PRIMARY key on a table, you must enter it as an element of the table you're working on. Check the syntax below:

{ CONSTRAINT (name of constraint) }

PRIMARY KEY < (name of column) {, (name of column)] … } >

SQL allows you to define primary keys using column constraints. However, you can only use this feature on a single column. Analyze the following example:

CREATE TABLE FICTION_NOVEL_AUTHORS

(AUTHOR_ID INT,

AUTHOR_NAME CHARACTER (50) PRIMARY KEY ,

PUBLISHER_ID INT) ;[15]

If you want to apply PRIMARY KEY on multiple columns (or store it as another definition), you may use it on the tabular level:

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)) ;

This approach places a primary key on two columns (i.e. AUTHOR_ID and AUTHOR_NAME). That means the paired values of the two columns need to be unique. However, duplicate values may exist inside any of the columns. Experienced database users

refer to this kind of primary key as a “superkey.” The term “superkey” means that the primary key exceeds the number of required columns.

In most cases, you need to set both UNIQUE and PRIMARY KEY constraints on a table. To achieve this, you just have to define the involved constraints, as usual. For instance, the code given below applies both of these constraints:

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

The following code will give you the same result:

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

[16]

The FOREIGN KEY Constraints

The constraints discussed so far focus on securing the data integrity of a table. NOT NULL stops columns from taking null values. PRIMARY KEY and UNIQUE, on the other hand, guarantee that the values of one or more columns are unique. In this regard, FOREIGN KEY (i.e. another SQL constraint) is different. FOREIGN

KEY, also called “referential constraint,” focuses on how information inside a table works with the information within another table.

This connection ensures information integrity throughout the database. In addition, the connection between different tables results to "referential integrity." This kind of integrity makes sure that data manipulation done on one table doesn’t affect the data inside other tables. The tables given below will help you understand this topic. Each of these tables, named PRODUCT_NAMES and PRODUCT_MANUFACTURERS, have one primary key:

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 PRODUCT_NAME_ID column of the PRODUCT_NAMES table has a PRIMARY KEY. The MANUFACTURER_ID of the

PRODUCT_MANUFACTURERS table has the same constraint. These columns are in yellow (see the tables above).

As you can see, the PRODUCT_NAMES table has a column called MANUFACTURER_ID. That column has the values of a column in the PRODUCT_MANUFACTURERS table. Actually, the MANUFACTURER_ID column of the PRODUCT_NAMES table can only accept values that come from the MANUFACTURER_ID column of the PRODUCT_MANUFACTURERS table.

Additionally, the changes that you'll make on the PRODUCT_NAMES table may affect the data stored in the PRODUCT_MANUFACTURERS table. If you remove a manufacturer, you also need to remove the entry from the MANUFACTURER_ID column of the PRODUCT_NAMES table. You can achieve this by using FOREIGN KEY. This constraint ensures the referential integrity of your database by preventing actions on any table from affecting the protected information.

Important Note: If a table has a foreign key, it is called “referencing table.” The table a foreign key points to is called “referenced table.”

When creating this kind of constraint, you need to obey the following guidelines:

You must define a referenced column by using PRIMARY KEY or UNIQUE. Most SQL programmers choose PRIMARY KEY for this purpose.

You may tag FOREIGN KEY constraints as column constraints or table constraints. You may work with any number of columns if you are using FOREIGN KEY as a table constraint. On the other hand, if you use this constraint at the column-level, you can only work on a single column.

A referencing table's foreign key should cover all of the columns you are trying to reference. In addition, the columns of the referencing table should match the data type of their counterparts (i.e. the columns being referenced). However, you don't have to use the same names for your referencing and referenced columns.

You don't need to indicate reference columns manually. If you don't specify any column for the constraint, SQL will consider the columns of the referenced table's primary key as the referenced columns. This process happens automatically.

You will understand these guidelines once you have analyzed the examples given below. For now, let's analyze the syntax of this constraint. Here's the format that you must use to apply FOREIGN KEY at the columnar level:

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

REFERENCES (name of the referenced table) { < (the referenced columns) > }

{ MATCH [SIMPLE | FULL | PARTIAL] }

{ (the referential action) }[17]

To use this FOREIGN KEY as a tabular constraint, you need to insert it as a table element. Here's the syntax:

{ 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) }[18]

You've probably noticed that FOREIGN KEY is more complex than the constraints you've seen so far. This complexity results from the constraint's option-filled syntax. However, generating this kind of constraint is easy and simple. Let's first analyze a basic example:

CREATE TABLE PRODUCT_NAMES

(PRODUCT_NAME_ID -> INT,

PRODUCT_NAME -> CHARACTER (50) ,

MANUFACTURER_ID -> INT -> REFERENCES PRODUCT_MANUFACTURERS) ;

This code applies the constraint on the MANUFACTURER_ID column. To apply this constraint on a table, you just have to type REFERENCES and indicate the referenced table's name. In addition, the columns of this foreign key are equal to that of the referenced table's primary key. If you don't want to reference your target's primary key, you need to specify the column/s you want to use. For instance, REFERENCES PRODUCT_MANUFACTURERS (MANUFACTURER_ID).

Important Note: The FOREIGN KEY constraint requires an existing referenced table. In addition, that table must have a PRIMARY KEY or UNIQUE constraint.

For the second example, you will use FOREIGN KEY as a tabular constraint. The code that you see below specifies the referenced column's name, even if that information is not required.

```
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) ) ;
```

[19]

You may consider the two lines at the bottom as the constraint's definition. The constraint's name, TS_MANUFACTURER_ID, comes after the keyword CONSTRAINT. You don't need to specify a name for your constraints since SQL will generate one for you in case this information is missing. On the other hand, you may want to set the name of your constraint manually since that value appears in errors (i.e. when SQL commands violate an existing

constraint). In addition, the names you will provide are more recognizable than system-generated ones.

Next, you should set the kind of constraint you want to use. Then, enter the name of your referencing column (MANUFACTURER_ID for the current example). You will then place the constraint on that column. If you are dealing with multiple columns, you must separate the names using commas. Afterward, type REFERENCES as well as the referenced table's name. Finally, enter the name of your referenced column.

That's it. Once you have defined this constraint, the MANUFACTURER_ID column of PRODUCT_NAMES won't take values except those that are already listed in the PRODUCT_MANUFACTURERS table's primary key. As you can see, a foreign key doesn't need to hold unique values. You may repeat the values inside your foreign keys as many times as you want, unless you placed the UNIQUE constraint on the column you're working on.

Now, let's apply this constraint on multiple columns. You should master this technique before studying the remaining elements of the constraint's syntax. For this example, let's use two tables: BOOK_AUTHORS and BOOK_GENRES.

The table named BOOK_AUTHORS has a primary key defined in the AUTHOR_NAME and AUTHOR_DOB columns. The SQL statement found below generates a table called BOOK_GENRES. This table has a foreign key consisting of the AUTHOR_DOB and DATE_OF_BIRTH columns.

```
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) ) ;[20]
```

This code has a pair of referenced columns (i.e. AUTHOR_NAME, AUTHOR_DOB) and a pair of referencing columns (i.e. AUTHOR_NAME and DATE_OF_BIRTH). The columns named AUTHOR_NAME inside the data tables contain the same type of data. The data type of the DATE_OF_BIRTH column is the same as that of AUTHOR_DOB. As this example shows, the name of a referenced column doesn't need to match that of its referencing counterpart.

The MATCH Part

Now, let's discuss another part of the constraint's syntax:

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

The curly brackets show that this clause is optional. The main function of this clause is to let you choose how to handle null values inside a foreign key column, considering the values that you may add to a referencing column. This clause won't work on columns that don't accept null values.

This part of the syntax offers three choices:

SIMPLE – If you choose this option, and at least one of your referencing columns has a null value, you may place any value on the rest of the referencing columns. The system will automatically trigger this option if you don't specify the MATCH section of your FOREIGN KEY's definition.

FULL – This option requires all of your referencing columns to accept null values; otherwise, none of them can accept a null value.

PARTIAL – With this option, you may place null values on your referencing columns if other referencing columns contain values that match their respective referenced columns.

The (referential action) Part

This is the final section of the FOREIGN KEY syntax. Just like the MATCH part, “referential action” is completely optional. You can use this clause to specify which actions to take when updating or removing information from one or more referenced columns.

For example, let's assume that you want to remove an entry from the primary key of a table. If a foreign key references the primary key you're working on, your desired action will violate the constraint. So, you should always include the data of your referencing columns inside your referenced columns.

When using this clause, you will set a specific action to the referencing table's definition. This action will occur once your referenced table gets changed.

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

According to this syntax, you may set ON DELETE, ON UPDATE, or both. These clauses can accept one of the following actions:

RESTRICT – This referential action prevents you from performing updates or deletions that can violate the FOREIGN KEY constraint. The information inside a referencing column cannot violate FOREIGN KEY.

SET NULL – This action changes the values of a referencing column to "null" if its corresponding referenced column gets removed or updated.

CASCADE – With this referential action, the changes you'll apply on a referenced column will also be applied to its referencing column.

NO ACTION – Just like RESTRICT, NO ACTION stops you from performing actions that will violate FOREIGN KEY. The main difference is that NO ACTION allows data violations while you are executing an SQL command. However, the information within your foreign key will not be violated once the command has been executed.

SET DEFAULT – With this option, you may set a referencing column to its default value by updating or deleting the data inside the corresponding referenced column. This referential action won't work if your referencing columns don't have default values.

To use this clause, you just have to insert it to the last part of a FOREIGN KEY's definition. Here's an example:

```
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

You can apply this constraint on a table, column, domain, or inside an assertion. This constraint lets you set which values to place inside your columns. You may use different conditions (e.g. value ranges) that define which values your columns may hold.

According to SQL programmers, the CHECK constraint is the most complex and flexible constraint currently available. However, this constraint has a simple syntax. To use CHECK as a column constraint, add the syntax below to your column definition:

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

If you want to use this constraint on a table, insert the syntax below to your table's definition:

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

Important Note: You'll later learn how to use this constraint on assertions and domains.

As this syntax shows, CHECK is easy to understand. However, its search condition may involve complex and extensive values. This constraint tests the assigned search condition for the SQL commands that try to alter the information inside a column protected by CHECK. If the result of the test is TRUE, the commands will run; if the result is false, the system will cancel the commands and display error messages.

You need to analyze examples in order to master this clause. However, almost all components of the search condition involve predicates. Predicates are expressions that work on values. In SQL, you may use a predicate to compare different values (e.g. COLUMN_3 < 5). The “less than” predicate checks whether the values inside COLUMN_3 are less than 5.

Most components of the search condition also utilize subqueries. Basically, subqueries are expressions that act as components of other expressions. You use a subquery if an expression needs to access or compute different layers of information. For instance, an expression might need to access TABLE_X to insert information to TABLE_Z.

In the example below, CHECK defines the highest and lowest values that you may enter in a column. This table definition generates a CHECK constraint and 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 )
 ) ;[21]
```

The resulting table will reject values that are outside the 1-50 range. Here's another way to write the 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 analyze the condition clause of these statements. This clause tells SQL that all of the values added to the STOCK_AVAILABILITY column must be lower than 50. The keyword AND informs SQL that there's another condition that must be applied. Finally, the clause tells SQL that each value added to

the said column should be higher than 1. To put it simply, each value should be lower than 50 and higher than 1.

This constraint also allows you to simply list your “acceptable values.” SQL users consider this a powerful option when it comes to values that won't be changed regularly. In the next example, you will use the CHECK constraint to define a book's genre:

```
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 ' ) ) ) ;[22]
```

Each value inside the GENRE column should be included in the listed genres of the condition. As you can see, this statement uses IN (i.e. an SQL operator). Basically, IN makes sure that the values within GENRE are included in the listed entries.

This constraint can be extremely confusing since it involves a lot of parentheses. You may simplify your SQL codes by dividing them into multiple lines. As an example, let's rewrite the code given above:

```
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 '
)
)
) ;
```

This style of writing SQL commands ensures code readability. Here, you need to indent the parentheses and their content so that they clearly show their position in the different layers of the SQL statement. By using this style, you can quickly identify the clauses placed in each pair of parentheses. Additionally, this statement works like the previous one. The only drawback of this style is that you need to use lots of spaces.

Let's analyze 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 ) ) ) ;
```

This code uses BETWEEN (i.e. another SQL operator) to set a range that includes the lowest and highest points. Because it has two ranges, it separates the range specifications using parentheses. The OR keyword connects the range specifications. Basically, OR tells SQL that one of the conditions need to be satisfied. Consequently, the values you enter in the column named

STOCK_AVAILABILITY should be from 1 through 50 or from 79 through 90.

How to Define an Assertion

Assertions are CHECK constraints that you can apply on multiple tables. Due to this, you can't create assertions while defining a table. Here's the syntax that you must use while creating an assertion:

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

Defining an assertion is similar to defining a table-level CHECK constraint. After typing CHECK, you need to specify the search condition/s.

Let's analyze a new example. Assume that the BOOK_TITLES table has a column that holds the quantity of books in stock. The total for this table should always be lower than your desired inventory. This example uses an assertion to check whether the total of the STOCK_AVAILABILITY column is lower than 3000.

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

This statement uses a subquery (i.e. “SELECT SUM (STOCK_AVAILABILITY) FROM BOOK_TITLES”) and compares it with 3000. The subquery starts with an SQL keyword, SELECT, which queries information from any table. The SQL function called SUM adds up all of the values inside STOCK_AVAILABILITY. The keyword FROM, on the other hand, sets the column that holds the table. The system will then compare the subquery's result to 3000. You will get an error message if you add an entry to the STOCK_AVAILABILITY column that makes the total exceed 3000.

How to Create a Domain and a Domain Constraint

As mentioned earlier, you may also insert the CHECK constraint into your domain definitions. This kind of constraint is similar to the ones you've seen earlier. The only difference is that you don't attach a domain constraint to a particular table or column. Actually, a domain constraint uses VALUE, another SQL keyword, while referring to a value inside a column specified for that domain. Now, let's discuss the syntax you need to use to generate new domains:

CREATE DOMAIN (name of domain) {AS } (type of data)

{ DEFAULT (the default value) }

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

This syntax has elements you've seen before, as you've seen default clauses and data types in the third chapter. The definition of the constraint, on the other hand, has some similarities with the ones discussed in the last couple of sections.

In the example below, you will generate an INT-type domain. This domain can only accept values between 1 and 50:

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

This example involves one new item, which is the VALUE keyword. As mentioned earlier, this keyword refers to a column's specified value using the BOOK_QUANTITY domain. Consequently, you will get an error message if you will enter a value that doesn't satisfy the assigned condition (i.e. each value must be between 1 and 50).

CHAPTER 4:How to Create an SQL View

“You can tell whether a man is clever by his answers… You can tell whether a man is wise by his questions.” – Naguib Mahfouz

Your database stores SQL information using “persistent” (i.e. permanent) tables. However, persistent tables can be impractical if you just want to check particular entries from one or more tables. Because of this, the SQL language allows you to use “views” (also called “viewed tables”).

Views are virtual tables whose definitions act as schema objects. The main difference between views and persistent tables is that the former doesn't store any data. Actually, viewed tables don't really exist – only their definition does. This definition lets you choose specific data from a table or a group of tables, according to the definition's query statements. To invoke a view, you just have to include its name in your query, as if it was an ordinary table.

How to Add a View to a Database

Views are extremely useful when you're trying to access various kinds of information. If you use a view, you may define complicated queries and save them inside a view definition. Rather than typing queries each time you use them, you may just call the view. In addition, views allow you to present data to other people without showing any unnecessary or confidential information.

For instance, you might need to allow some users to access certain parts of employee records. However, you don't want the said users to access the SSN (i.e. social security number) or pay rates of the listed employees. Here, you may generate views that show only the data needed by users.

How to Define an SQL View

In SQL, the most basic view that you can create is one which points to a single table and collects information from columns without changing anything. Here is the basic syntax of a view:

CREATE VIEW (name of view) { < (name of the view's columns) > }

AS (the query)

{ WITH CHECK OPTION }

Important Note: This part of the book focuses on the first and second lines of the format. You'll learn about the third line later.

You need to set the view's name in the first part of the definition. Additionally, you should name the view's columns if you are facing any of the following circumstances:

If you need to perform an operation to get the column's values, instead of just copying them from a table.

If you are working with duplicate column names. This situation happens when you combine tables.

You may set names for your columns even if you don't need to. For instance, you may assign logical names to your columns so that even an inexperienced user can understand them.

The second part of the format has a mandatory keyword (i.e. AS) and a placeholder for the query. Despite its apparent simplicity, the query placeholder may involve a complicated structure of SQL statements that perform different operations.

Let's analyze a basic example:

CREATE VIEW BOOKS_IN_STOCK

(BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY) AS

SELECT BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY

FROM BOOK_INVENTORY[23]

This sample is one of the simplest views that you can create. It gets three columns from a table. Remember that SQL isn't strict when it comes to line breaks and spaces. For instance, while creating a view, you may list the column names (if applicable) on a separate line. Database management systems won't care which coding technique you use. However, you can ensure the readability of your codes by adopting a coding style.

Now, let's dissect the sample code given above. The first part sets BOOKS_IN_STOCK as the view's name. The second part sets the name of the columns and contains the SQL keyword AS.

If you don't specify the names you want to use, the view's columns will just copy the names of the table's columns. The last two lines hold the search expression, which is a SELECT statement. Here it is:

SELECT BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY

FROM BOOK_INVENTORY

SELECT is flexible and extensive: it allows you to write complex queries that give you the exact kind of information you need.

The SELECT statement of this example is basic. It only has two clauses: SELECT and FROM. The first clause sets the column to be returned. The second clause, however, sets the table where the information will be pulled from. Once you call the BOOKS_IN_STOCKS view, you will actually call the embedded SELECT command of the view. This action gets the information from the correct table/s.

For the second example, let’s create a view that has an extra 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;

The last clause sets a criterion that should be satisfied for the system to retrieve data. The only difference is that, rather than pulling the authors' information, it filters search results based on the year each book was published.

Important Note: The contents of the last clause don't affect the source table in any way. They work only on the information returned by the view.

You may use WHERE in your SELECT statements to set different types of criteria. For instance, you can use this clause to combine tables. Check the following code:

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;[24]

This code creates a view named BOOK_PUBLISHERS. The BOOK_PUBLISHERS view contains two columns: BOOK_TITLE and PUBLISHER_NAME. With this view, you'll get data from two different sources: (1) the BOOK_TITLE column of the

BOOK_INVENTORY table and (2) the PUBLISHER_NAME column of the TABS table.

For now, let's focus on the third clause (i.e. the SELECT statement). This clause qualifies the columns based on the name of their respective tables (e.g. BOOK_INVENTORY .BOOK_TITLE). If you are joining tables, you need to specify the name of each table to avoid confusion. Obviously, columns can be highly confusing if they have duplicate names. However, if you're dealing with simple column names, you may omit the name of your tables. For instance, your SELECT clause might look like this:

SELECT BOOK_TITLE, PUBLISHER_NAME

Now, let's discuss the statement's FROM section. When combining tables, you need to name all of the tables you want to use and separate the entries using commas. Aside from the concern regarding duplicate names, this clause is identical to that of previous examples.

WHERE, the last clause of this statement, matches data rows together. This clause is important since, if you don't use it, you won't be able to match values you've gathered from different tables. In the current example, the values inside the TAG_ID column of BOOK_INVENTORY should match the values inside the TAG_ID column of the table named TAGS.

SQL allows you to qualify a query by expanding the latter'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;
```
[25]

Let's work on another example. Similar to the examples you've seen earlier, this view collects information from a single table. This view, however, performs computations that return the modified information. Here is the statement:

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

This statement creates a view that has three columns: BOOK_TITLE, ORIGINAL_PRICE, and REDUCED_PRICE. Here, SELECT indicates the columns that hold the needed information. The statement defines BOOK_TITLE and ORIGINAL_PRICE using the methods discussed in the previous examples. The system will copy the data inside the BOOK_INVENTORY table's BOOK_TITLE and ORIGINAL_PRICE columns. Then, the system will paste the data to the columns of the same name inside the BOOK_DISCOUNTS view.

However, the last column is different,. Aside from taking values from its corresponding column, it multiplies the collected values by 0.8 (i.e. 80%). This way, the system will determine the correct values to display in the view's REDUCED_PRICE column.

SQL also allows you to insert the WHERE clause to your SELECT statements. Here's an example:

```
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;[26]
```

This WHERE clause limits the search to entries whose STOCK_AVAILABILITY value is higher than 20. As this example shows, you may perform comparisons on columns that are included in the view.

How to Create an Updateable View

In the SQL language, some views allow you to perform updates. Simply put, you may use a view to alter the information (i.e. add new rows and/or alter existing information) inside the table you're working on. The “updateability” of a view depends on its SELECT statement. Usually, views that involve simple SELECT statements have higher chances of becoming updateable.

Remember that SQL doesn't have syntax to create updateable views. Rather, you need to write a SELECT statement that adheres to certain standards. This is the only way for you to create an updateable view.

The examples you've seen in this chapter imply that the SELECT statement serves as the search expression of a CREATE VIEW command. To be precise, query expressions may belong to different kinds of expressions. As an SQL user, most of the time, you'll be dealing with query specifications. Query expressions are SQL

expressions that start with SELECT and contain different elements. To keep it simple, let's assume that SELECT is a query specification. Database products also use this assumption, so it is certainly effective.

You can't summarize, combine, or automatically delete the information inside the view.

The table you're working with should have at least one updateable column.

Every column inside the view should point to a single column in a table.

Every row inside the view should point to a single row in a table.

How to Drop a View

In some cases, you need to delete a view from a database. To do that, you need to use the following syntax:

DROP VIEW (name of the view);

The system will delete the view as soon as you run this statement. However, the process won't affect the underlying information (i.e. the data stored inside the actual tables). After dropping a view, you may recreate it or use its name to generate another view. Let's analyze a basic example:

DROP VIEW BOOK_PUBLISHERS;

This command will delete the BOOK_PUBLISHERS view from the database. However, the underlying information will be unaffected.

Database Security

Security is an important element of every database. You need to make sure that your database is safe from unauthorized users who may view or alter data. Meanwhile, you also need to ensure that authorized users can access and/or change data without any

problems. The best solution for this problem is to provide each user with the privileges they need to do their job.

To protect databases, SQL has a security scheme that lets you specify which database users can view specific information from. This scheme also allows you to set what actions each user can perform. This security scheme (or model) relies on authorization identifiers. As you've learned in the second chapter, authorization identifiers are objects that represent one or more users that can access/modify the information inside the database.

The Security Model of SQL

The security of your database relies on authorization identifiers. You can use these identifiers to allow other people to access and/or alter your database entries. If an authorization identifier lacks the right privileges to alter a certain object, the user won't be able to change the information inside that object. Additionally, you may configure each identifier with various kinds of privileges.

In SQL, an authorization identifier can be a user identifier (i.e. “user”) or a role name (i.e. “role”). A “user” is a security profile that may represent a program, a person, or a service. SQL doesn't have specific rules regarding the creation of a user. You may tie the identifier to the OS (i.e. operating system) where the database system runs. Alternatively, you may create user identifiers inside the database system itself.

A role is a group of access rights that you may assign to users or other roles. If a certain role has access to an object, all users you've assigned that role to can access the said object.

SQL users often utilize roles to provide uniform sets of access rights to other authorization identifiers. One of the main benefits offered by a role is that it can exist without any user identifier.

That means you can create a role before creating a user. In addition, a role will stay in the database even if you have deleted all of your user identifiers. This functionality allows you to implement a flexible process to administer access rights.

The SQL language has a special authorization identifier called PUBLIC. This identifier covers all of the database users. Similar to other identifiers, you may assign access rights to a PUBLIC profile.

Important Note: You need to be careful when assigning access rights to the PUBLIC identifier, as users might use that identifier for unauthorized purposes.

Creating and Deleting a Role

Generating new roles is a simple process. The syntax has two clauses: an optional clause and a mandatory clause.

CREATE ROLE (name of role)

{ WITH ADMIN [CURRENT_ROLE | CURRENT_USER] }

As you can see, CREATE ROLE is the only mandatory section of this statement. You don't need to set the statement's WITH ADMIN part. Actually, SQL users rarely set that clause. WITH ADMIN becomes important only if your current role name/user identifier pair doesn't have any null values.

Let's use the syntax to create a role:

CREATE ROLE READERS;

That's it. After creating this role, you will be able to grant it to users or other roles.

To drop (or delete) a role, you just have to use the following syntax:

DROP ROLE (name of role)

This syntax has a single requirement: the name of the role you want to delete.

Granting and Revoking a Privilege

Whenever you grant a privilege, you are actually linking a privilege to an authorization identifier. You will place this privilege/authorization identifier pair on an object, allowing the former to access the latter based on the defined privileges.

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] }

This syntax has three mandatory clauses, namely: ON, TO and GRANT. The last two clauses, GRANTED BY and WITH GRANT OPTION, are completely optional.

The process of revoking privileges is simple and easy. You just have to use the syntax given below:

REVOKE { GRANT OPTION FOR } [(list of privileges) | ALL PRIVILEGES]

ON (type of object) name of object)

FROM [{ list of authorization identifiers) | PUBLIC }

CHAPTER 5: Database Creation

"To improve is to change; to be perfect is to change often." – Sir Winston Churchill

Data is stored in tables and indexed to make queries more efficient. Before you can create tables, you need a database to hold the table. If you're starting from zero, you will have to learn to create and use a database.

Creating a Database

"We are what we pretend to be, so we must be careful about what we pretend to be." – Kurt Vonnegut

To create a database, you will use the CREATE command with the name of the database.

The following is the syntax:

CREATE DATABASE database_name;

To demonstrate, assume you wanted to create a database and name it xyzcompany:

CREATE DATABASE xyzcompany;

With that statement, you have just created the xyzcompany database. Before you can use this database, you need to designate it as the active database. You have to run the USE command with the database name to activate your new database.

Here's the statement:

USE xyzcompany;

In following sessions, you can just type the statement 'USE xyzcompany' to access the database.

Removing a Database

If you need to remove an existing database, you can easily do so with this syntax:

DROPDATABASE databasename:

Therefore, you must exercise caution when using the DROP command to remove a database. You will also need admin privileges to drop a database.

Schema Creation

The CREATE SCHEMA statement is used to define a schema. On the same statement, you can also create objects and grant privileges on these objects.

The CREATE SCHEMA command can be embedded within an application program. Likewise, it can be issued using dynamic SQL statements. For example, if you have database admin privileges, you can issue this statement which creates a schema called USER1 with the USER1 as its owner:

CREATE SCHEMA USER1 AUTHORIZATION USER1

The following statement creates a schema with an inventory table. It also grants authority on the inventory 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

MySQL 5.7

If you're using MySQL 5.7, CREATE SCHEMA is synonymous to the CREATE DATABASE command.

Here's the syntax:[27]

CREATE {DATABASE | SCHEM A}[IF NOT EXISTS] db_name [create_specification]…

create_specification:

[DEFAULT] CHARACTER SET [=] charset_name | [DEFAULT] COLLATE [=] collation_name

Oracle 11g

In Oracle 11g, you can create several views and tables and perform several grants in one transaction within the CREATE SCHEMA statement. To successfully execute the CREATE SCHEMA command, Oracle runs each statement within the block and commits the transaction if no errors are encountered. If a statement returns an error, all statements are rolled back.

The statements CREATE VIEW, CREATE TABLE, and GRANT may be included within the CREATE SCHEMA statement. Hence, you must not only have the privilege to create a schema, you must also have the privileges needed to issue the statements within it.

The syntax is the following:[28]

{ create_table_statement

| create_view_statement

| grant_statement

}…;

SQL Server 2014

In SQL Server 2014, the CREATE SCHEMA statement is used to create a schema in the current database. This transaction may also create views and tables within the newly-created schema and set GRANT, REVOKE, or DENY permission on these objects.

This statement creates a schema and sets the specifications for each argument:[29]

CREATE SCHEM A schem a_name_clause [<schem a_element> […n]]

<schem a_name_clause>::=

{

schem a_name

| AUTHORIZATION owner_name

| scheme a_name AUTHORIZATION owner_name

}

<schem a_element> ::=

{

table_definition | view_definition | grant_statement |

revoke_statement |deny_statement

}

PostgreSQL 9.3.13

The CREATE SCHEMA statement is used to enter a new schema into a database. The schema name should be unique within the current database.[30]

Here's the syntax:

CREATE SCHEM A schem a_name [AUTHORIZATION user_name] [schem a_element[…]]

CREATE SCHEM A AUTHORIZATION user_name [schem a_element[…]]

CREATE SCHEM A IF NOT EXISTS schem a_name [AUTHORIZATION user_name]

CREATE SCHEM A IF NOT EXISTS AUTHORIZATION user_name

Creating Tables and Inserting Data Into Tables

Tables are the main storage of information in databases. Creating a table means specifying a name for a table, defining its columns, as well as the data type of each column.

How to Create a Table

The keyword CREATE TABLE is used to create a new table. It is followed by a unique identifier and a list that defines each column and the type of data it will hold.

CREATE TABLE table_name

(

colum n1 datatype [NULL | NOT NULL].

colum n2 datatype [NULL | NOT NULL].

…

):

This is the basic syntax to create a table:

Parameters

table_name: This is the identifier for the table.

column1, column2: These are the columns that you want the table to have. All columns should have a data type. A column is defined as either NULL or NOT NULL. If this is not specified, the database will assume that it is NULL.

The following set of questions can serve as a guide when creating a new table:

- What is the most appropriate name for this table?
- What data types will I be working with?
- What is the most appropriate name for each column?
- Which column(s) should be used as the main key(s)?

- What type of data can be assigned to each column?
- What is the maximum width for each column?
- Which columns can be empty and which columns should not be empty?

The following example creates a new table with the xyzcompany database. It will be named EMPLOYEES:

```
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)
);
```

The code creates a table with 6 columns. The ID field was specified as its primary key. The first column is an INT data type with a precision of 6. It does not accept a NULL value. The second column, FIRST_NAME, is a VARCHAR type with a maximum range of 35 characters. The third column, LAST_NAME, is another VARCHAR type with a maximum of 35 characters. The fourth column, POSITION, is a VARCHAR type which is set at a maximum of 35 characters. The fifth column, SALARY, is a DECIMAL type with a precision of 9 and scale of 2. Finally, the fifth column, ADDRESS, is a VARCHAR type with a maximum of 50 characters. The id column was designated as the primary key.

Creating a New Table Based on Existing Tables

You can create a new table based on an existing table by using the CREATE TABLE keyword with SELECT.

Here's the syntax:[31]

```
CREATE TABLE new_table_name AS
(
SELECT [colum n1.column, column n, 2'''colum nN]
FROM existing_table_name
[WHERE]
);
```

Executing this code will create a new table with column definitions that are identical to the original table. You may copy all columns or select specific columns for the new table. The new table will be populated by the values of the original table.

To demonstrate, create a duplicate table named STOCKHOLDERS from the existing EMPLOYEES table within the xyzcomppany. Here's the code:

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

Inserting Data into Table

SQL's Data Manipulation Language (DML) is used to perform changes to databases. You can use DML clauses to fill a table with fresh data and update an existing table.

Populating a Table with New Data

There are two ways to fill a table with new information: manual entry or automated entry through a computer program.

Populating data manually involves data entry using a keyboard, while automated entry involves loading data from an external source. It may also include transferring data from one database to a target database.

Unlike SQL keywords or clauses that are case-insensitive, data is case-sensitive. Hence, you have to ensure consistency when using or referring to data. For instance, if you store an employee's first name as 'Martin', you should always refer to it in the future as 'Martin' and never 'MARTIN' or 'martin'.

The INSERT Keyword

The INSERT keyword is used to add records to a table. It inserts new rows of data to an existing table.

There are two ways to add data with the INSERT keyword. In the first format, you simply provide the values for each field and they will be assigned sequentially to the table's columns. This form is generally used if you need to add data to all columns.

You will use the following syntax for the first form:

INSERT INTO table_name

VALUES ('value1', 'value2', [NULL];

In the second form, you'll have to include the column names. The values will be assigned based on the order of the columns' appearance. This form is typically used if you want to add records to specific columns.

Here's the syntax:

INSERT INTO table_name (colum n1, colum n2, colum n3)

VALUES ('value1', 'value2', 'value3');

Notice that in both forms, a comma is used to separate the columns and the values. In addition, you have to enclose character/string and

date/time data within quotation marks.

Assuming that you have the following record for an employee:

First NameRobert

Last NamePage

PositionClerk

Salary5,000.00

282 Patterson Avenue Illinois

To insert this data into the EMPLOYEES table, you can use the following statement:

INSERT INTO EMPLOYEES (FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS)

VALUES (‘Robert’, ‘Page’, ‘Clerk’, 5000.00, ‘282 Patterson Avenue, Illinois’);

To view the updated table, here’s the syntax:

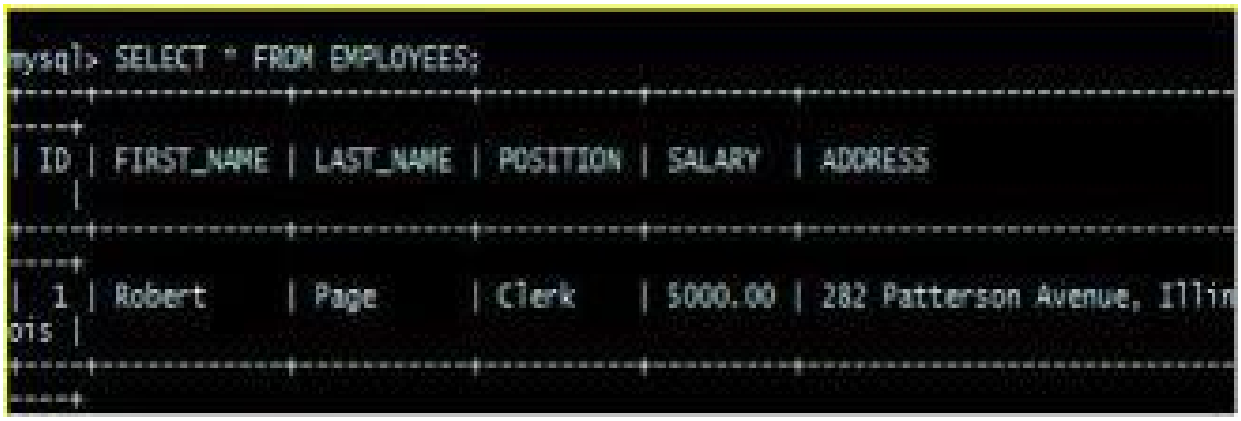
SELECT * FROM table_name;

To display the data stored in the EMPLOYEES’ table, use the following statement:

SELECT * FROM EMPLOYEES:

The wildcard (*) character tells the database system to select all fields on the table.

Here’s a screenshot of the result:[32]



Now, try to encode the following data for another set of employees:

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

You will have to repeatedly use the INSERT INTO keyword to enter each employee data to the database. Here's how the statements would look:

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’);

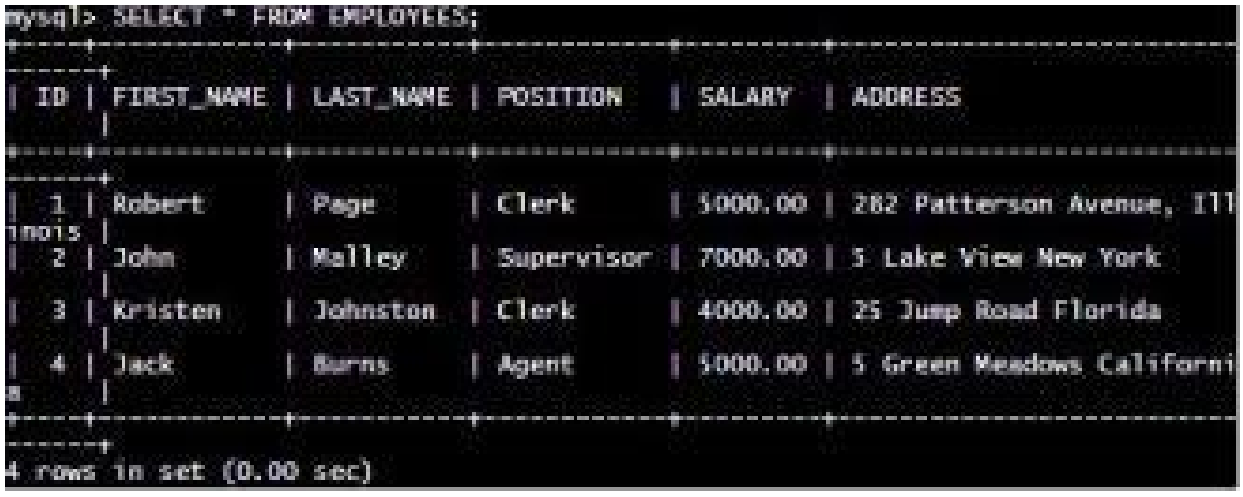
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS)

VALUES(‘Jack’, ‘Burns’, ‘Agent’, 5000.00, ‘5 Green Meadows, California’)’;

To fetch the updated EMPLOYEES table, use the SELECT command with the wild card character.

SELECT * FROM EMPLOYEES:

Here’s a screenshot of the result:



Notice that SQL assigned an ID number for each set of data you entered. This is because you have defined the ID column with the auto_increment attribute. This property will prevent you from using the first form when inserting data. So, you have to specify the rest of the columns in the INSERT INTO table_name line.[33]

Inserting Data into Specific Columns

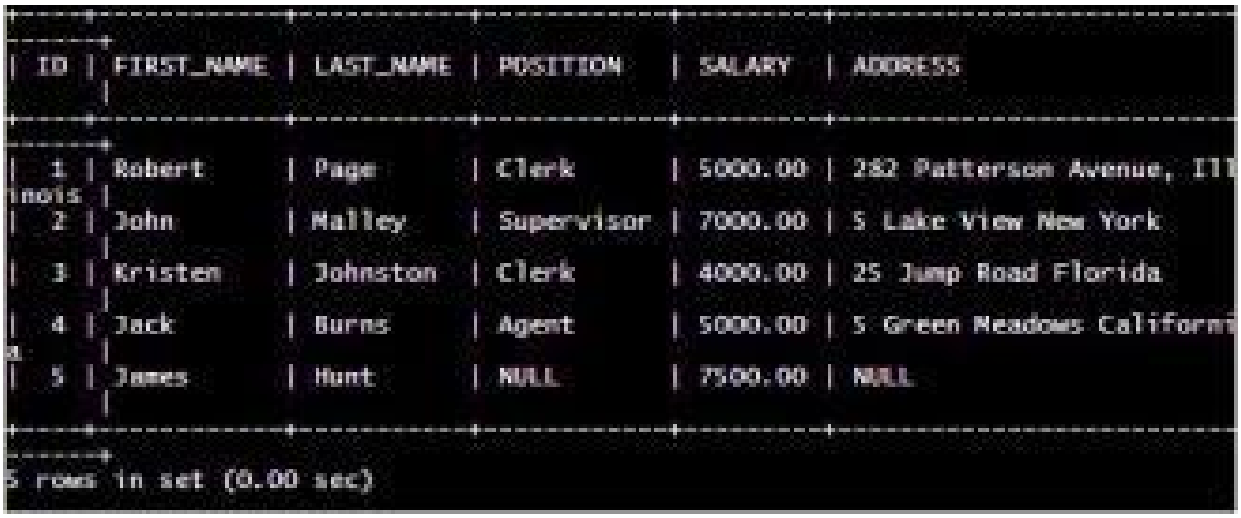
You may also insert data into specific column(s). You can do this by specifying the column name inside the column list and the corresponding values inside the VALUES list of the INSERT INTO statement. For example, if you just want to enter an employee’s full

name and position, you will need to specify the column names FIRST_NAME, LAST_NAME, and SALARY in the columns list and the values for the first name, last name, and salary inside the VALUES list.

To see how this works, try entering the following data into the EMPLOYEES table:

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

Here's a screenshot of the updated EMPLOYEES table:[34]



```
+----+------------+-----------+------------+---------+-----------------------------
------+
| ID | FIRST_NAME | LAST_NAME | POSITION   | SALARY  | ADDRESS
     |
+----+------------+-----------+------------+---------+-----------------------------
------+
|  1 | Robert     | Page      | Clerk      | 5000.00 | 282 Patterson Avenue, Ill
inois |
|  2 | John       | Malley    | Supervisor | 7000.00 | 5 Lake View New York
     |
|  3 | Kristen    | Johnston  | Clerk      | 4000.00 | 25 Jump Road Florida
     |
|  4 | Jack       | Burns     | Agent      | 5000.00 | 5 Green Meadows Californi
a    |
|  5 | James      | Hunt      | NULL       | 7500.00 | NULL
     |
+----+------------+-----------+------------+---------+-----------------------------
------+
5 rows in set (0.00 sec)
```

Inserting NULL Values

In some instances, you may have to enter NULL values into a column. For example, you may not have data on hand to enter a new employee's salary. It may be misleading to provide just about any salary figure.

Here's the syntax:

INSERT INTO schem a.table_name

VALUES ('colum n1', NULL, 'colum n3);

You'll need this answer sheet to be able to check your syntax in each exercise to ensure that it is correct. You are more than welcome to use it if you're stuck on an exercise as well. At the end of each exercise, I encourage you to check your answers.

Each exercise gives an overview, as well as examples, before you start applying what you learned in each section.

Below is a sample of the Product table that you will be using in the next few exercises.

Product ID	Name	Product Number	Color	Standard Cost	List Price	Size
A **317**	A LL Crankarm	A CA-5965	A Black			

A 0	A 0	A NULL				
318	ML Crankarm	CA-6738	Black	0	0	NULL
A **319**	A HL Crankarm	A CA-7457	A Black	A 0	A 0	A NULL
320	Chainring Bolts	CB-2903	Silver	0	0	NULL
A **321**	A Chainring Nut	A CN-6137	A Silver	A 0	A 0	A NULL

Query Structure and SELECT Statement

Understanding the syntax and its structure will be extremely helpful for you in the future. Let's delve into the basics of one of the most common statements, the SELECT statement, which is used solely to retrieve data.

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

In the above query, you're selecting two columns and all rows from the entire table. Since you're selecting two columns, the query will perform much faster than if you had selected all of the columns like this:

```
SELECT *
FROM Table_Name
```

Select all of the columns from the Production.Product table.

Using the same table, select only the ProductID, Name, Product Number, Color and Safety Stock Level columns.

*(**Hint**: The column names do not contain spaces in the table!)*

The WHERE Clause

The WHERE clause is used to filter the amount of rows returned by the query. This clause works by using a condition, such as if a column is equal to, greater than, less than, a certain value.

When writing your syntax, it's important to remember that the WHERE condition comes after the FROM statement. The types of operators that can be used vary based on the type of data that you're filtering.

EQUALS – This is used to find an exact match. The syntax below uses the WHERE clause for a column that contains the exact string of 'Value'. Note that strings need single quotes around them, but numerical values do not.

```
SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 = 'Value'
```

BETWEEN – Typically used to find values between a certain range of numbers or date ranges. It's important to note that the first value in the BETWEEN comparison operator should be lower than the value on the right. Note the example below comparing between 10 and 100.

```
SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 BETWEEN 100 AND 1000
```

GREATER THAN, LESS THAN, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO – SQL Server has comparison operators that you can use to compare certain values.

```
SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 =< 1000 --Note that you can also use <, >, >= or even <> for not equal to
```

LIKE – This searches for a value or string that is contained within the column. You would still use single quotes, but include the percent

symbols indicating if the string is in the beginning, at the end or anywhere between ranges of strings.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 LIKE '%Value%' --Searches for the word 'value' in any field

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 LIKE 'Value%' --Searches for the word 'value' at the beginning of the field

SELECT Column_Name1, Column_Name2

FROM Table_Name [35]

WHERE Column_Name3 LIKE '%Value' --Searches for the word 'value' at the end of the field

IS NULL and IS NOT NULL – As previously discussed, NULL is an empty cell that contains no data. Eventually, you'll work with a table that does contain NULL values, which you can handle in a few ways.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 IS NULL --Filters any value that is NULL in that column, but don't put NULL in single quotes since it's not considered a string.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 IS NOT NULL --Filters any value that is not NULL in that column.

Using the Production.Product table again, select the Product ID, Name, Product Number and Color. Filter the products that are silver colored.

Using ORDER BY

The ORDER BY clause is simply used to sort data in ascending or descending order, and is specified by which column you want to sort. This command also sorts in ascending order by default, so if you want to sort in descending order, use DESC in your command.

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 = 'Value'
ORDER BY Column_Name1, Column_Name2 --Sorts in ascending order, but you can also include ASC to sort in ascending order.
```

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
WHERE Column_Name3 = 'Value'
ORDER BY Column_Name1, Column_Name2 DESC --This sorts the data in descending order by specifying DESC after the column list.
```

Select the Product ID, Name, Product Number and List Price from the Production.Product table where the List Price is between $100.00 and $400.00. Then, sort the results by the list price from smallest to largest.

*(**Hint**: You won't use the $ in your query and also, you may refer back to the sorting options you just reviewed if you need to.)*[36]

Data Definition Language (DDL)

"There is nothing worse than aggressive stupidity." – Johann Wolfgang von Goethe

DDL is the SQL syntax that is used to create, alter or remove objects within the instance or database itself. Below are some examples to help you get started. DDL is split up into three types of commands:

CREATE - This will create objects within the database or instance, such as other databases, tables, views, etc.

--Creates a Database

CREATE **DATABASE** DatabaseName

--Creates a schema (or container) for tables in the current database

CREATE SCHEMA SchemaName

--Creates a table within the specified schema

CREATE TABLE SchemaName.TableName

(

Column1 datatype PRIMARY KEY,

Column2 datatype(n),

Column3 datatype

)[37]

--Creates a View

CREATE VIEW ViewName

AS

SELECT

Column1,

Column2

FROM TableName

ALTER - This command will allow you to alter existing objects, like adding an additional column to a table or changing the name of the database, for example.

--Alters the name of the database

ALTER DATABASE DatabaseName MODIFY NAME = NewDatabaseName

--Alters a table by adding a column

ALTER TABLE TableName

ADD ColumnName datatype(n)[38]

DROP - This command will allow you to drop objects within the database or the database itself. This can be used to drop tables, triggers, views, stored procedures, etc. Please note that these items will not exist within your database anymore – or the database will cease to exist if you drop it.

--Drops a Database - use the master db first

USE master

GO

--Then drop the desired database

DROP DATABASE DatabaseName

GO

--Drops a table; should be performed in the database where the specified table exists

DROP TABLE Table_Name

--Drops a view from the current database

DROP VIEW ViewName

Applying DDL Statements

Open a new query window on your own machine by clicking the "New Query" button and ensure that you're using the 'master' database.

You can tell which database you're using and switch between databases by checking below and using the drop-down menu.

Back in the Guidelines for Writing T-SQL Syntax section, you learned about the 'USE' statement and splitting up batches of code with 'GO'.

You'll be using this in the next exercise. So, feel free to refer back to that section before or during the exercise.

Create a database called Company. Don't forget to use the GO statement between using the master database and creating your new database!

After you've performed this, you'll now have a database that you can begin adding tables to. Below are a series of exercises in which you'll begin creating your first schema and table within this database.

Create a schema called Dept. The table that you'll be creating in the next exercise will be associated to this schema.

Create a table called Department. The table should have two columns: Department_Id, which should be an integer data type and the primary key. The other column should be Department_Name, which should be a data type of VARCHAR of 30 bytes.

Alter the name of your database from Company to Company_Db.

Running the DDL Script

Now, it's time for you to download the following DDL script to create the rest of the tables that you'll be working with. Don't worry, I've written the script so that it will successfully run by clicking the

‘Execute’ button. Just make sure that you have completed the exercises correctly, otherwise it will not work.

Once you’ve performed this, you’ll start using your knowledge of DML to add and retrieve data.

Data Manipulation Language (DML)

DML or Data Manipulation Language is exactly what it sounds like; it manipulates the data. The statements that are part of DML allow you to modify existing data, create new data, or even delete data.

Take note that DML is not specific to objects like modifying a table’s structure or a database’s settings, but it works with the data within such objects.

SELECT - This statement is the most common SQL statement, and it allows you to retrieve data. It is also used in many reporting scenarios.

--Selects two columns from a table

SELECT

Column_Name1,

Column_Name2

FROM Table_Name

INSERT - This statement allows you to insert data into the tables of a particular database.

--Inserts three rows into the table

INSERT INTO Table_Name

(Column_Name1, Column_Name2) --These two items are the columns

VALUES

('Value1', 1), --Here, I'm inserting Value1 into Column_Name1 and the number 1 in Column_Name2.

('Value2', 2),

('Value3', 3)

Now, it is time to insert data into your Department table. Insert the values as shown below. For example, the Department Name of Marketing must have a Department ID of 1, and so on.

Department_Id: 1, Department_Name: Marketing

Department_Id: 2, Department_Name: Development

Department_Id: 3, Department_Name: Sales

Department_Id: 4, Department_Name: Human Resources

Department_Id: 5, Department_Name: Customer Support

Department_Id: 6, Department_Name: Project Management

Department_Id: 7, Department_Name: Information Technology

Department_Id: 8, Department_Name: Finance/Payroll

Department_Id: 9, Department_Name: Research and Development

UPDATE - Allows you to update existing data. For instance, you may change an existing piece of data in a table to another value.

It’s recommended to use caution when updating values in a table. You could give all rows within one column the same value if you don’t specify a condition in the WHERE clause.

--Updating a value or set of values in one column of a table

UPDATE Table_Name

SET

Column_Name1 = 'New Value 1', --Specify what your new value should be here

Column_Name2 = 'New Value 2'

WHERE

Column_Name1 = 'Old Value' --Using the WHERE clause as a condition

--Updates all rows in the table with the same value for one specific column

UPDATE Table_Name

SET Column_Name1 = 'Value'[39]

Now, you will update one of the department names in your table. Let's say that you feel like the name 'Finance/Payroll' won't work because payroll is handled by employees via an online web portal. So, that leaves finance, but it's best to call this department 'Accounting' instead, due to the responsibilities of those in the department.

Update the Department table and change the value from 'Finance/Payroll' to 'Accounting' based on its ID.

DELETE - This action is self-explanatory; it deletes data from a specified object, like a table. It's best to use caution when running the DELETE statement.

If you don't use the WHERE clause in your query, you'll end up deleting all of the data within the table. So, it's always best to use a WHERE clause in a DELETE statement.

*Note: Much like the UPDATE statement, you can also swap out DELETE with SELECT in your statement to see what data you will be deleting, as long as you're using the WHERE clause in your query. Like the UPDATE statement, when you delete one value, it's best to use the primary key as a condition in your WHERE clause!

--Uses a condition in the DELETE statement to only delete a certain value or set of values

DELETE FROM Table_Name

WHERE Column_Name1 = 'Some Value'

--Deletes all of the data within a table since the WHERE clause isn't used

DELETE FROM Table_Name

Let’s say that this company doesn’t actually have a Research and Development (R&D) department. Perhaps they haven’t grown enough yet or don’t require that department.

For this exercise, delete the ‘Research and Development’ department from the Department table based on its ID.

Running the DML Script

Now, it’s time for you to download the following DML script in order to populate the Company_Db database with data. Again, make sure that you have completed the exercises correctly, otherwise it will not work.

This data will be used in the following exercises in the ‘Transforming Data’ section. So, please keep in mind that each exercise refers to the Company_Db database, unless specified otherwise.

CHAPTER 6: Database Administration

"Laws too gentle are seldom obeyed; too severe, seldom executed." – Benjamin Franklin

In order to start on the path of a Database Administrator, Database Developer or even a Data Analyst, you'll need to know how to back up, restore and administer databases. These are essential components to maintaining a database and are definitely important responsibilities.

Recovery Models

There are several different recovery models that can be used for each database in the SQL Server. A recovery model is an option or model for the database that determines the type of information that can be backed up and restored.

Depending on your situation, like if you cannot afford to lose critical data or you want to mirror a Production environment, you can set the recovery model to what you need. Also, keep in mind that the recovery model that you choose could also affect the time it takes to back up or restore the database and logs, depending on their size.

Below are the recovery models and a brief explanation of each. The SQL statement example at the end of each is what you would use in order to set the recovery model.

Full

This model covers all of the data within the database, as well as the transaction log history.

When using this model and performing a restore of the database, it offers "point in time" recovery. This means that if there was a point where data was lost from the database during a restore for example, it allows you to roll back and recover that data.

This is the desired recovery model if you cannot afford to lose any data.

It may take more time to back up and restore, depending on the size of the data.

- Can perform full, differential and transaction log backups.

ALTER DATABASE DatabaseName SET RECOVERY FULL

Simple

This model covers all of the data within the database too, but recycles the transaction log. When the database is restored, the transaction log will be empty and will log new activity moving forward.

Unlike the “Full” recovery model, the transaction log will be reused for new transactions and therefore cannot be rolled back to recover data if it has accidentally been lost or deleted.

It’s a great option if you have a blank server and need to restore a database fairly quickly and easily, or to just mirror another environment and use it as a test instance.

- Can perform full and differential backups only, since the transaction log is recycled per this recovery model.

ALTER DATABASE DatabaseName SET RECOVERY SIMPLE

Bulk Logged

This particular model works by forgoing the bulk operations in SQL Server, like BULK INSERT, SELECT INTO, etc. and does not store these in the transaction logs. This will help free up your transaction logs and make it easier and quicker during the backup and restore process.

Using this method, you have the ability to use “point in time” recovery, as it works much like the “Full” recovery model.

If your data is critical to you and your database processes often use bulk statements, this recovery model is ideal. However, if your database does not often use bulk statements, the “Full” recovery model is recommended.

- Can perform full, differential and transaction log backups.

ALTER DATABASE DatabaseName SET RECOVERY BULK_LOGGED

Let’s say you’re in the middle of a large database project and need to back up the database. Use SQL syntax (and the examples above) to set the recovery model for Company_Db to “Full” to prepare for the backup process.

Database Backup Methods

There are a few main backup methods that can be used to back up databases in SQL Server, and each one is dependent on the recovery model being used. In the previous section regarding recovery models, the last bullet point in each model discussed the types of backups that can be performed, i.e. full, differential and transaction log.

Note: *if you don’t remember the types of backups that can be performed for each recovery model, just give yourself time! Eventually, you will remember them!*

Each backup method will perform the following action:

Full:

Backs up the database in its entirety, as well as the transaction log (if applicable.)

Ideal if the same database needs to be added to a new server for testing.

It may take a longer period of time to back up if the database is large in size, in addition to it backing up the entire database and transaction log.

- However, this doesn't need to be performed nearly as often as a differential since you're backing up the entire database.

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

Differential:

It is dependent upon a full back up, so the full backup must be performed first. However, this will back up any data changes between the last full back up and the time that the differential backup takes place.

It is ideal in any situation, but must be done more frequently since there is new data being added to the database frequently.

It is much faster to back up than a full backup and takes up less storage space.

- However, this doesn't capture the database entirely, so this is the reason it must be performed periodically.

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

Transaction Log:

Like a differential backup, it's dependent upon a full backup.

Backs up all of the activity that has happened in the database.

Useful and necessary in some cases, like when restoring a database in full.

Since they hold the activity within the database's history, it's wise to back this up frequently so that the logs do not grow large in size.

Some logs are initially larger in size, but can become smaller as more frequent backups are done (depends on the activity within the database).

- Therefore, the log could be quick to back up and restore, if the database has minimal activity and is backed up frequently.

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

In the last exercise, you set the recovery model of the Company_Db to "Full". Now for this exercise, you'll be backing up the database.

First, back up the database in full using the full backup method.

Next, delete all of the data from the Sales.Product_Sales table (intended to be a simulation of losing data.)

Finally, query the table that you just deleted data from to ensure that the data doesn't exist. Later on, you'll be restoring the database and ensuring that the data is the Sales.Product_Sales table.

Database Restores

In the database world, you will probably hear that the most important part of restoring a database is using a solid backup. That's entirely true! This is why it's important to ensure that you're using the proper recovery model.

Preparing to Restore the Database

An important thing to note is that the .bak file typically contains file groups or files within them. Each file could either be a full or differential backup. It's important to know which one you're

restoring, but you can also specify which file(s) you'd like to restore. However, if you don't specify, SQL Server will pick the default file, which is 1. This is a typical full back up.

First, use RESTORE HEADERONLY to identify the files within your .BAK file. This is something you should do prior to restoring a database.

--View database backup files

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

Once you run the above query, look for the number in the "BackupType" column. The 1 indicates a full backup, whereas a 5 indicates a differential backup.

Also, the file number that you need to use is in the "Position" column. In the example below, note that the file number 8 is a full backup and file number 9 is a differential backup.

Results | Messages

	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

Additionally, you must first use the 'master' database and then set the desired database to a non-active state (SINGLE_USER) to only allow one open connection to the database prior to restoring.

In the below syntax, the WITH ROLLBACK IMMEDIATE means that any incomplete transactions will be rolled back (not completed) in order to set the database to a single, open connection state.

--Sets the database to allow only one open connection

--All other connections to the database must be closed, otherwise the restore fails

ALTER DATABASE DatabaseName SET SINGLE_USER WITH ROLLBACK IMMEDIATE

Once the restore has completed, you can set the database back to an active state (MULTI_USER), as shown below. This state allows multiple users to connect to the database, rather than just one.

--Sets the database to allow multiple connections

ALTER DATABASE DatabaseName SET MULTI_USER

Database Restore Types

Below is an overview of the types of restores that you can perform depending on your situation/needs.

Full Restore

Restores the entire database, including its files.

Overwrites the database if it already exists using the WITH REPLACE option.

If the recovery model is set to “Full”, you need to use the WITH REPLACE option.

Use the FILE = parameter to choose which file you’d like to restore (can be full or differential).

If the recovery model is set to “Simple”, you don’t need the WITH REPLACE option.

If the database does not exist, it will create the database, including its files and data.

Restores from a .BAK file.

--If the recovery model is set to Full - also choosing the file # 1 in the backup set

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

--If recovery model is set to Simple

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

Differential Restore

Use RESTORE HEADERONLY in order to see the backup files you have available, i.e. the full and differential backup file types.

You must perform a full restore of the .BAK file first with the NORECOVERY OPTION, as the NORECOVERY option indicates other files need to be restored as well.

Once you've specified the full backup file to be restored and use WITH NORECOVERY, you can restore the differential.

Finally, you should include the RECOVERY option in your last differential file restore in order to indicate that there are no further files to be restored.

As a reference, the syntax below uses the files from the previous screenshot of RESTORE HEADERONLY.

--Performs a restore of the full database backup file first

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

--Now performs a restore of the differential backup file

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

Log Restore

This is the last step to perform a thorough database restore.

You must restore a “Full” or “Differential” copy of your database first, then restore the log.

- Finally, you must use the NORECOVERY option when restoring your database backup(s) so that your database stays in a restoring state and allows you to restore the transaction log.

Since you already performed a backup of your database and “lost” data in the Sales.Product_Sales table, it’s now time to restore the database in full.

In order to do this, you’ll first need to use the ‘master’ database, then set your database to a single user state, perform the restore, and finally set it back to a multi-user state.

Once you’ve restored the database, retrieve all data from the Sales.Product_Sales table to verify that the data exists again.

Attaching and Detaching Databases

As you know, because you already went through this portion, the methodology of attaching and detaching databases is similar to backups and restores.

Essentially, here are the details of this method:

Allows you to copy the .MDF file and .LDF file to a new disk or server.

Performs like a backup and restore process, but can be faster at times, depending on the situation.

- The database is taken offline and cannot be accessed by any users or applications. It will remain offline until it's been reattached.

So, which one should you choose? Though a backup is the ideal option, there are cases where an attachment/detachment of the database may be your only choice.

Consider the following scenarios:

Your database contains many file groups. Attaching that can be quite cumbersome.

The best solution would be to back up the database and then restore it to the desired destination, as it will group all of the files together in the backup process.

Based on the size of the database, the backup/restore process takes a long time. However, the attaching/detaching of the database could be much quicker if it's needed as soon as possible.

- In this scenario, you can take the database offline, detach it, and re-attach to the new destination.

As you know, there are two main file groups when following the method of attaching databases. These files are .MDF and .LDF. The .MDF file is the database's primary data file, which holds its structure and data. The .LDF file holds the transactional logging activity and history.

However, a .BAK file that's created when backing up a database groups all of the files together and you restore different file versions from a single backup set.

Consider your situation before using either option, but also consider a backup and restore to be your first option, then look into the attach/detach method as your next option. Also, be sure to test it before you move forward with live data!

Attaching/Detatching the AdventureWorks2012 Database

Since you already attached this database, let's now detach it from the server. After that, you'll attach it again using SQL syntax.

Detaching the Database

In SQL Server, there's a stored procedure that will detach the database for you. This particular stored procedure resides in the 'master' database. Under the hood, you can see the complexity of the stored procedure by doing the following:

Click to expand the Databases folder

Click on System Databases, then the Master database

Click on Programmability

Click on Stored Procedures, then System Stored Procedures

- Find sys.sp_detach_db, right-click it and select 'Modify' in SSMS. You'll then see its syntax.

For this, simply execute the stored procedure as is.

The syntax is the following:

```
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 expand a little on what is happening, you want to use the 'master' database to alter the database you'll be detaching and set it

to single user instead of multi-user.

Finally, the value after dbname allows you to specify the name of the database to be detached, and the skipchecks being set to false means that the database engine will update the statistics information, identifying that the database has been detached. It's ideal to set this as false whenever detaching a database so that the system holds current information about all databases.

Now, detach the AdventureWorks2012 database from your server instance. Use the above SQL example for some guidance.

Attaching Databases

Once you have detached your database, if you navigate to where your data directory is, you'll see that the AdventureWorks2012_Data.MDF file still exists – and it should since you only detached it and haven't deleted it.

Next, take the file path of the .MDF file and copy and paste it in a place that you can easily access, like on the notepad.

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

Once you've connected to your instance and opened up a new query session, you will just need to use the path where the data file is stored. Once you have that, you can enter that value in the following SQL syntax examples in order to attach your database.

Below is the syntax used to attach database files and log files. So, in the following exercise, you'll skip attaching the log file completely, since you're not attaching this to a new server. So, you may omit the statement to attach the log file.

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

In the above example, I am calling out the statement to attach the log file if one is available. However, if you happen not to have the .LDF file but only the .MDF file, that's alright. You can just attach the .MDF file and the database engine will create a new log file and start writing activity to that particular log.

For this exercise, imagine that you've had 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.

Each exercise gives an overview, as well as examples, before you start applying what you learn in each section.

Below is a sample of the Production.Product table that you'll be using in the next couple of exercises.

Product ID	Name	Product Number	Color	Standard Cost	List Price	Size
317	LL Crankarm	CA-5965	Black	0	0	NULL
318	ML Crankarm	CA-6738	Black	0	0	NULL
319	HL Crankarm	CA-7457	Black	0	0	NULL
320	Chainring Bolts	CB-2903	Silver	0	0	NULL
321	Chainring Nut	CN-6137	Silver	0	0	NULL

Query Structure and SELECT Statement

Understanding the syntax and its structure will be extremely helpful for you in the future. Let's delve into the basics of one of the

most common statements, the SELECT statement, which is used solely to retrieve data.

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

In the above query, you're selecting two columns and all rows from the entire table. Since you're selecting two columns, the query performs much faster than if you had selected all of the columns like this:

```
SELECT *
FROM Table_Name
```

Select all of the columns from the table Production.Product.

Using the same table, select only the ProductID, Name, Product Number, Color and Safety Stock Level columns.

*(**Hint**: The column names do not contain spaces in the table!)*

The WHERE Clause

The WHERE clause is used to filter the amount of rows returned by the query. This clause works by using a condition, such as if a column is equal to, greater than, less than, between, or like a certain value.

When writing your syntax, it's important to remember that the WHERE condition comes after the FROM statement.

EQUALS – This is used to find an exact match. The syntax below uses the WHERE clause for a column that contains the exact string of 'Value'. Note that strings need single quotes around them, while numerical values do not.

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

WHERE Column_Name3 = 'Value'

BETWEEN – Typically used to find values between a certain range of numbers or date ranges. It's important to note that the first value in the BETWEEN comparison operator should be lower than the value on the right. Note the example below comparing between 10 and 100.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 BETWEEN 10 AND 100

GREATER THAN, LESS THAN, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO – SQL Server has comparison operators that you can use to compare certain values.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 =< 1000 --Note that you can also use <, >, >= or even <> for not equal to

LIKE – This searches for a value or string that is contained within the column. You still use single quotes but include the percent symbols indicating if the string is in the beginning, end or anywhere between ranges of strings.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 LIKE '%Value%' --Searches for the word 'value' in any field

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 LIKE 'Value%' --Searches for the word 'value' at the beginning of the field

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 LIKE '%Value' --Searches for the word 'value' at the end of the field

IS NULL and IS NOT NULL – As previously discussed, NULL is an empty cell that contains no data. Eventually, you'll work with a table that does contain NULL values, which you can handle in a few ways.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 IS NULL --Filters any value that is NULL in that column, but don't put NULL in single quotes since it's not considered a string.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 IS NOT NULL --Filters any value that is not NULL in that column. [40]

Using the Production.Product table again, select the Product ID, Name, Product Number and Color. Filter the products that are silver colored.

Using ORDER BY

The ORDER BY clause is simply used to sort data in ascending or descending order, specific to which column you want to start. This command also sorts in ascending order by default, so if you want to sort in descending order, use DESC in your command.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 = 'Value'

ORDER BY Column_Name1, Column_Name2 --Sorts in descending order, but you can also include ASC to sort in ascending order.

SELECT Column_Name1, Column_Name2

FROM Table_Name

WHERE Column_Name3 = 'Value'

ORDER BY Column_Name1, Column_Name2 DESC --This sorts the data in descending order by specifying DESC after the column list.

Then, sort the results in the list price from smallest to largest.

*(**Hint**: You won't use the $ in your query and if needed, you can refer back to the sorting options you just reviewed.)*

CHAPTER 7: SQL Transaction

"Just because something doesn't do what you planned it to do doesn't mean it's useless." – Thomas Edison

What is an SQL Transaction?

Transactions are a sequence of tasks performed in a logical order against a database. We consider each transaction as a single unit of work. Transactions give you more control over your SQL behavior, which becomes essential if you are continuing to maintain your data integrity, or planning to avoid database errors.

In Chapter 3, we talked at length about data integrity and the various methods one can adopt to ensure the presence of such integrity. However, that takes into account the fact that somehow, multiple users will not have access to the same data.

In reality, this may not be the case.

In a situation where many users are trying to modify the data, there are chance occurrences of actions overlapping each other. In many instances, one user takes specific operations on data that may not be valid. She or he does not realize that the data has been compromised because another user took action at the same time. As neither user will be immediately aware of any form of change taking place, the data might continue to remain inconsistent for quite a while.

With this in mind, all users will continue to assume the data is intact and that there are no problems for them to oversee.

Let us try to understand the above scenario with an example. Assume that there are two users, User A, and User B, who are both working on a customer's data at the same time.

User A might notice that the customer's last name is incorrect, and will set about changing the name, immediately saving the data after making the change. User B might be working on another section of the customer's data, such as the customer's address or telephone number. However, User A's changes might not reflect on User B's side. User B might still be working on the customer's data using the old, and wrong, last time, inadvertently changing the last name to the incorrect entry when saving the data.

Now both users are unaware of the inconsistencies in their work, and this problem might go unnoticed for a long while. It might seem somewhat trivial when contemplating the fact that only a single error took place. However, imagine the occurrences repeating themselves overtime, creating inconsistencies on numerous data. When the time comes, you will be forced to look through all the data you have. If your database is a large one, this might take a considerable amount of your time.

With SQL transactions, consecutive actions or changes will not affect the validity of data that is seen by another user.

SQL transactions give you the power to commit SQL statements, which will apply said statements into the database. Alternatively, you have the ability to rollback, which will undo statements from the database.

In an SQL transaction, an action will convert into a transaction if it successfully passes four characteristics, denoted by the acronym ACID.

Atomic

This property focuses on the "everything or nothing" principle of a transaction. It ensures the performance of all the operations in a transaction. If an operation fails any point, the entire transaction they will be aborted. Additionally, if only a few statements are

executed, the transactions are rolled back to their previous state. To complete a transaction successfully and to apply it to the database, all operations should be correct and none should be missing.

Consistent

With this property, the database must not merely be consistent at the beginning of the transaction, but after its completion as well. Hence, if a set of operations result in actions that change the consistency at any point of their performance, then the transaction will be reverted to its original state.

Isolated

When a particular transaction is taking place, the data might be inconsistent. This inconsistency has the potential to affect the database and hence until the transaction is completed, the data will not be available for other operations. It will be free from any outside effect. In other others, it will be isolated. Additionally, no other transaction can influence or affect the isolated transaction in any way, ensuring that there is no compromise in integrity.

Durable

When all transactions are completed, any changes or modifications must be preserved and maintained. In the event of hardware or application error, there should be no loss of data; it should be available reliably and consistently. This characteristic ensures that your data is available in the event of such

One has to note that at any time, whether operations are completed or reverted, the transaction maintains the integrity of the database.

There are seven commands that you can use to manage transactions. These are:

COMMIT: Saves changes and commits them to the database.

ROLLBACK: Rolls back changes to a specific SAVEPOINT, or the beginning of a transaction. Changes are not applied to the database.

SAVEPOINT: Creates a savepoint – a form of marker – within a transaction. When you ROLLBACK, you can reach this point instead of the beginning of the transaction.

RELEASE SAVEPOINT: Removes or releases a savepoint. After this command, any ROLLBACK will revert the transaction to its original state.

SET TRANSACTION: Creates characteristics for the execution of a particular transaction.

START TRANSACTION: Sets the characteristics of the transactions and begins the transaction.

SET CONSTRAINTS: Establishes the constraint mode within a transaction. A constraint creates a condition where a decision is passed to either apply the constraint as soon as modifications on the data begin to occur or delay the application of the constraint until the completion of the transaction.

Time for another example.

It is important to remember that when you are about to start a transaction, the database is in its original state. At this state, you can expect the data to be consistent. Once you execute the transactions, the SQL statements within those transactions begin processing. If the process is successful, then one uses the COMMIT command. This command causes the SQL statements to update the database and dismiss the transaction. If the process of updating is not successful, then the ROLLBACK command comes into play. This process gives you a brief overview of commands. However, we will be going in-depth into their functions.

Commit

When all the statements are accomplished in a specific transaction, then the next step would be to terminate the transaction. One of the recommended ways to terminate a transaction is to commit all the changes to the database made so far. When you are confident of the changes you have performed so far, you can then utilize the COMMIT command. See the below syntax statements that you can use to commit a statement:

COMMIT [WORK] [AND CHAIN]

COMMIT [WORK] [AND NO CHAIN]

You must remember that the usage of the WORK entry is not mandatory. Only the COMMIT keyword will suffice to process the COMMIT command. In actuality, the COMMIT and COMMIT WORK entries have the same purpose. Earlier versions of SQL used to have the inclusion of the WORK keyword and some users might be more familiar with it. However, if you are not used to its usage, then you do not have to include it to execute the COMMIT command.

The AND CHAIN article is another optional inclusion in the COMMIT statement. It is not a commonly used statement in SQL implementations. The AND CHAIN clause simply tells the system to begin with a new transaction as soon as the current one ends. With this clause, you will have no need to use the SET TRANSACTION or START TRANSACTION statements, leaving the work to the system to continue to the next step automatically.

However, this might prove to be a disadvantage rather than a convenience. As you will need complete control over your transaction, it is best to use an alternative. Rather than using AND CHAIN in your COMMIT statement, try to utilize the AND NO CHAIN parameter. This essentially lets your system know that it should not begin a new transaction with the same settings are the

current one. In the case where there is no mention of either of the parameters (AND CHAIN or AND NO CHAIN), the AND NO CHAIN will be the default clause the system will adopt.

Your COMMIT statement might look like the below:

```
COMMIT;
```

We have not included any of the two clauses mentioned. However, know that if you would like the system to start a new transaction automatically, use the below statement:

```
COMMIT AND CHAIN;
```

Assume you have a table named EMPLOYEES with “names” and “salaries” columns.

NAME	SALARIES
John	5,000
Mary	7,000
Jennifer	9,000
Mark	10,000
Adam	7,000

If you would like to remove salaries that are 7,000, then you simply have to use the below:

```
SQL > DELETE FROM EMPLOYEES
    WHERE SALARIES = 7,000
SQL > COMMIT;
```

Now, as mentioned before, if you would like the next transaction to begin after the execution of the current one, then your statement will look something like this:

```
SQL > DELETE FROM EMPLOYEES
    WHERE SALARIES = 7,000
SQL > COMMIT AND CHAIN;
```

Rollback

Human error is a permanent fixture in any process. For this reason, if you find a situation that calls for a roll back, then this statement gives you the power to commit such an action.

By using ROLLBACK, you will undo the statements you have done so far, either bringing them back to a specific savepoint – if you had established such a savepoint – or sending them to the beginning of the transaction.

Let us look at some of the parameters or inclusions of a ROLLBACK statement:

```
ROLLBACK [WORK] [AND CHAIN]
ROLLBACK [WORK] [AND NO CHAIN]
```

When a user makes changes to data but does not use the COMMIT statement, then those changes are stored in a temporary format called a transaction log. Users can look at this unsaved data, analyzing it to check if it meets particular requirements. If the changes do no express the desired result users are aiming for, they can hit ROLLBACK, ensuring that the data is not saved, and they can work on it again.

Users can also use the ROLLBACK feature to send the database to a savepoint or the beginning of a transaction in the event of a hardware malfunction or application crash. If a power loss

interrupted your work, then as soon as the system restarts, the ROLLBACK feature will review all pending transactions and will roll back all statements.

Unlike the COMMIT statement, ROLLBACK has the option to include a TO SAVEPOINT parameter. When using the TO SAVEPOINT clause, the system will not cancel the transaction. You will merely be taken to a specific point from where you can continue your work.

Should you wish to cancel the transaction simply remove the TO STATEMENT parameter from your statement.

Here is an example of the most fundamental form of a ROLLBACK statement:

```
ROLLBACK;
```

Notice that we have not used the AND CHAIN parameter. This is because, as mentioned before in the section for COMMIT command, the system uses the AND NO CHAIN command by default. Alternatively, you can use the AND CHAIN command should you wish to initiate a new transaction after the conclusion of the current one. However, bear in mind that you cannot use the TO SAVEPOINT and AND CHAIN parameters at the same time. This is because a TO SAVEPOINT command requires the termination of the current transaction.

When including the TO SAVEPOINT parameter, ensure that you add the name of the savepoint. See an example of this process below:

```
ROLLBACK TO SAVEPOINT;
```

The SAVEPOINT value is replaced by the name of the savepoint as shown in the example below:

```
BEGIN
    SELECT customer_number, customer_lastname, purchases
```

```
    DELETE FROM customer_number WHERE purchases < 10000;
    SAVEPOINT section_1;
    ROLLBACK TO section_1;
END;
```

While the above example is a rather simple form of the command, its purpose is to give you clarity on SAVEPOINT usages and naming. Do note that you can enter as many parameters between the SAVEPOINT command and the ROLLBACK clause.

If you specify a savepoint name, then the rollback command will take you back to that particular savepoint, irrespective of the fact that there could be other savepoints in between.

```
BEGIN
SELECT customer_number, customer_lastname, purchases
DELETE FROM customer_number WHERE purchases < 10000;
SAVEPOINT section_1;
SAVEPOINT section_2;
SAVEPOINT section_3;
SAVEPOINT section_4;
ROLLBACK TO section_1;
END;
```

Notice that even though you have SAVEPOINTS named from section_2 to section_4, you have asked the system to roll back to section_1. By doing so, the system will ignore all of the other savepoints.

Savepoint

While we are on the subject of savepoints, let us look at why these commands are important, and how to work with them.

Essentially, you will be working with a complex set of transactions. To make it easier to understand the transaction, you ill group different sections into units. Breaking down transactions in this matter will allow you to identify problems easily and know which section to reach in order to solve the issue.

But the question arises: how can you reach these sections without having to change the entire transaction?

Why, you use the SAVEPOINT command, of course!

At this point, you should be aware of an important point. MySQL and Oracle support the SAVEPOINT parameter, but if you are working with SQL Server, you might be using the SAVE TRANSACTION query instead.

However, the functions of both SAVEPOINT and SAVE TRANSACTION parameters are the same.

Let us look at an example to understand how a savepoint function works.

STEP 1: Start Transaction

STEP 2: SQL Statements

STEP 3: If the SQL statement is successful, the commence execution. If the SQL statement is not successful, then roll back to the beginning of the transaction

STEP 4: Enter SAVEPOINT 1

STEP 5: SQL Statements

STEP 6: If the SQL statement is successful, the commence execution. If the SQL statement is not successful, then roll back to SAVEPOINT 1

STEP 7: Enter SAVEPOINT 2

STEP 8: SQL Statements

STEP 9: If the SQL statement is successful, the commence execution. If the SQL statement is not successful, then roll back to SAVEPOINT 2

STEP 10: COMMIT

In the above example, we set savepoints after working on SQL statements. When we perform this routine, we begin to work on specific SQL statements before jumping on to the next one. This ensures that we execute all statements properly, and if there is an error in the statements, we can rework them. Once the system deems statements fully successful, we can use a savepoint and move on to the next set of operations. By doing this, we save the integrity of the first group of operations. After the savepoint, we can continue our progress, knowing that there will not be any changes occurring to the first statements.

This becomes a convenient method of dealing with a set of actions without worrying about the integrity of previous actions.

The process of creating a savepoint is rather simple. You just have to use the following command:

SAVEPOINT <name of the savepoint>

The savepoint name does not have to be “SECTION_1”. That was the name chosen for the purpose of the example. You can select your own name, preferably something you can easily recollect when you want to.

Release savepoint

After you have completed certain operations within a transaction, you might not require the savepoints you had previously established. In order to remove them, you can use the RELEASE SAVEPOINT command.

However, do note that once you release a savepoint, you will not be able to roll back to it again. Which is why the ideal way to release a savepoint is to check if you are satisfied with your work so far. Due diligence might be an added task, but it will provide you with the capability to complete your work in confidence.

To release a savepoint, simply enter the below command:

RELEASE SAVEPOINT <name of the savepoint>

Let us take an example where you have the below savepoints:

SAVEPOINT section_1;

SAVEPOINT section_2;

SAVEPOINT section_3;

SAVEPOINT section_4;

In order to release all of them, you will have to specify each one.

Hence, the process is as shown below:

RELEASE SAVEPOINT section_1;

RELEASE SAVEPOINT section_2;

RELEASE SAVEPOINT section_3;

RELEASE SAVEPOINT section_4;

You might release all savepoints at one time or you might choose to release one of the savepoints while keeping the rest. This is acceptable. All you have to ensure is that you use the right savepoint name. Another note to remember is that you do not have to use the release command in the order of the savepoints you have created. You can choose to release any savepoint at any time, and in any order.

Set Transaction

This command allows you to work with the different properties of a transaction. These could be one of the below:

- Read Only
- Read Write
- Specify an isolation level
- Attach it to a rollback property

Any effect of the SET TRANSACTION operation affects only your transaction. Other users will not notice any changes on their end.

You can establish SET TRANSACTION using the below command:

SET TRANSACTION <mode>

In the above command, the <mode> refers to the type of option you would like to specify. You can use three modes in SET TRANSACTION. These are:

- Access Level
- Isolation Level
- Diagnostics Size

You can add multiple transaction modes into the <mode> placeholder. If you wish to do that, separate the different modes by a comma. However, you cannot repeat the same mode again.

Example: you can include an isolation level and work on the diagnostics size. However, you cannot include two diagnostics size mode.

Now let us try to look at each mode separately.

Access Level

In a SET TRANSACTION, there are two types of access levels; READ ONLY and READ WRITE. If you select the READ ONLY option, the system will not allow you to make any changes to the

database. However, if you choose the READ WRITE access level, you will be able to add statements in your transactions to modify the data or even the database structure.

Isolation Level

Do you remember how we talked about isolating transactions so that nothing can influence it while you are working on it? Well, here we will talk about setting up isolation levels.

For a SET TRANSACTION, you can use four isolation levels, as seen in the syntax below:

SET TRANSACTION ISOLATED LEVEL

READ UNCOMMITTED

READ COMMITTED

REPEATABLE READ

SERIALIZABLE

The isolation levels are arranged in the order of their effectiveness, with the READ UNCOMMITTED being the least level of isolation you can use and the SERIALIZABLE being the highest level of isolation. If you do not specify the level of isolation, the system will assume the SERIALIZATION level, giving your transaction maximum isolation.

Diagnostic Size

The SET TRANSACTION allows you to specify a diagnostic size. The size you include lets the system know how much area to allow you for conditions. In an SQL statement, a condition is a message, warning, or other notifications raised by the statement. If you do not specify a diagnostic size, then your database will automatically assign you one. This number is not fixed. It varies from database to database.

Start Transaction

In a database, you can start a transaction explicitly or automatically when you begin executing a command.

For example, you can start a transaction when you use commands such as DELETE, CREATE TABLE, and so on.

Alternatively, you can commence a transaction by using the START TRANSACTION statement.

As with the SET TRANSACTION, you can specific one of more modes here as well. These are the access level, isolation level, and diagnostic size modes that we had mentioned before.

Here is an example of a START TRANSACTION command:

```
START TRANSACTION
    READ ONLY
    INSOLATION LEVEL SERIALIZED
    DIAGNOSTICS SIZE 10;
```

Once you enter the syntax above, the START TRANSACTION will execute the statement and its operations.

Set Constraint

When you are working with transactions, you might encounter scenarios where your work will go against the constraints established in the database. Let us take an example.

Assume that you have a table that includes the NO NULL constraint. Now, while you are working on the table, you realize that you might not have a value to place instead of NO NULL constraint at that point in time. But you cannot leave without entering a value. This forces you to insert random values into the section.

To avoid such situations, you can define a constraint as deferrable.

The syntax for this action is shown below:

SET CONSTRAINTS (constraint names) DEFERRED/IMMEDIATE

To mention multiple constraint names, you need to separate them by a bar. For example:

(constraint_1 | constraint_2 | constraint_3)

However, if you wish to choose all constraints – and this could be useful when you have lots of them in your database – then you simply have to use the word ALL

Here is an example:

SET CONSTRAINTS ALL DEFERRED/IMMEDIATE

Another thing to note is that you do not have to mention both DEFERRED and IMMEDIATE. You have to choose the one that fits your activity at that point.

For example, if you are working on the database, you will choose to defer the constraints, in which case your statement should look like this:

SET CONSTRAINTS ALL DEFERRED

Once you have completed operations on the database, you can then choose to apply the changes. In that case, the statement will look like this:

SET CONSTRAINT ALL IMMEDIATE

So through this chapter, you have understood the idea behind SQL transactions, and why they are essential when working with SQL programming.

CHAPTER 8:Logins, Users and Roles

“See first that the design is wise and just; that ascertained, pursue it resolutely.” – William Shakespeare

Server Logins

Server logins are user accounts that are created in SQL servers and use SQL Server authentication, which entails entering a password and username to log into the SQL Server instance. SQL Server authentication is different from Windows Authentication, as Windows doesn’t prompt you to enter your password.

These logins can be granted access to be able to log in to the SQL Server, but don't necessarily have database access depending on permissions that are assigned. They can also be assigned certain server level roles that provide certain server permissions within SQL Server.

Server Level Roles

Permissions are wrapped into logins by Server Level Roles. The Server Level Roles determine the types of permissions a user has.

There are nine types of predefined server level roles:

Sysadmin - can perform any action on the server, essentially the highest level of access.

Serveradmin - can change server configurations and shutdown the SQL server.

Securityadmin - can GRANT, DENY and REVOKE server level and database level permissions, if the user has access to the database. The securityadmin can also reset SQL Server passwords.

Processadmin - provides the ability to end processes that are running in a SQL Server instance.

Setupadmin - can add or remove linked servers by using SQL commands.

Diskadmin - this role is used to manage disk files in the SQL Server.

Dbcreator - provides the ability to use DDL statements, like CREATE, ALTER, DROP and even RESTORE databases.

Public - all logins are part of the ‘public’ role and inherit the permissions of the ‘public’ role if it does not have specific server or database object permissions.

If you really want to delve deeper into this topic (by all means, I encourage you to do so), then go ahead and click this link to get more information on server roles via the Microsoft website.

Below is an example of syntax that can be used to create server logins. You are able to swap out ‘dbcreator’ with any of the roles above when you assign a server level role to your login.

You'll also notice the brackets, [], around ‘master' and ‘User Name' for instance. These are delimited identifiers, which we have previously discussed.

```
--First, use the master database
USE [master]
GO
--Creating the login - replace 'User A' with the login name that you'd like to use
--Enter a password in the 'Password' field - it's recommended to use a strong password
--Also providing it a default database of 'master'
CREATE LOGIN [User A] WITH PASSWORD= N'123456',
DEFAULT_DATABASE=[master]
```

GO[41]

Create a user with a login name called Server User. Give this user a simple password of 123456, and give it default database access to ‘master’.

Assigning Server Roles

There are a few different ways to assign these roles to user logins in the SQL Server. One of the options is to use the interface in Management Studio, and the other is to use SQL syntax.

Below is the syntax to give a user a particular server role. If you recall the previous syntax we used to create the login for ‘User A’, this will give ‘User A’ a server role of dbcreator.

--Giving the User a login a role of 'dbcreator', (it has the public role by default)

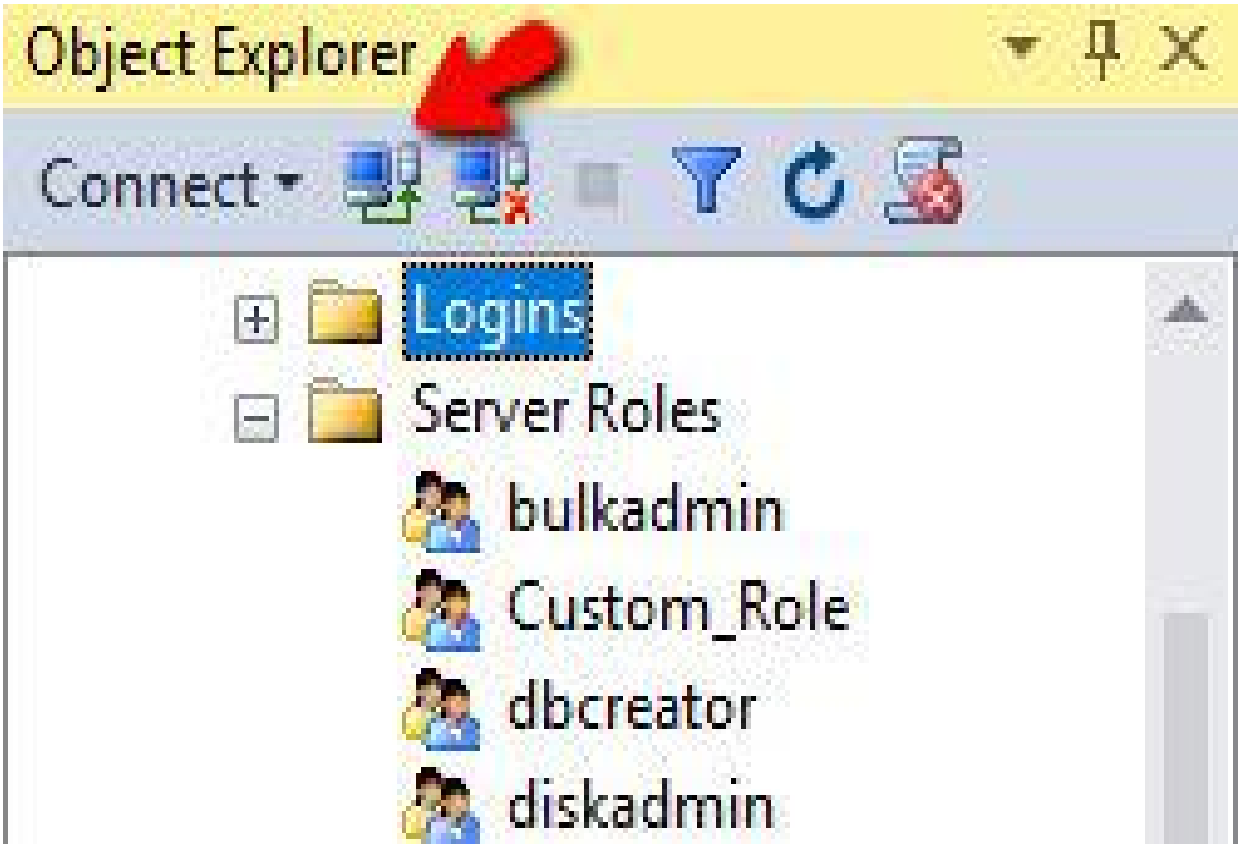
ALTER SERVER ROLE dbcreator ADD MEMBER [User A]

GO

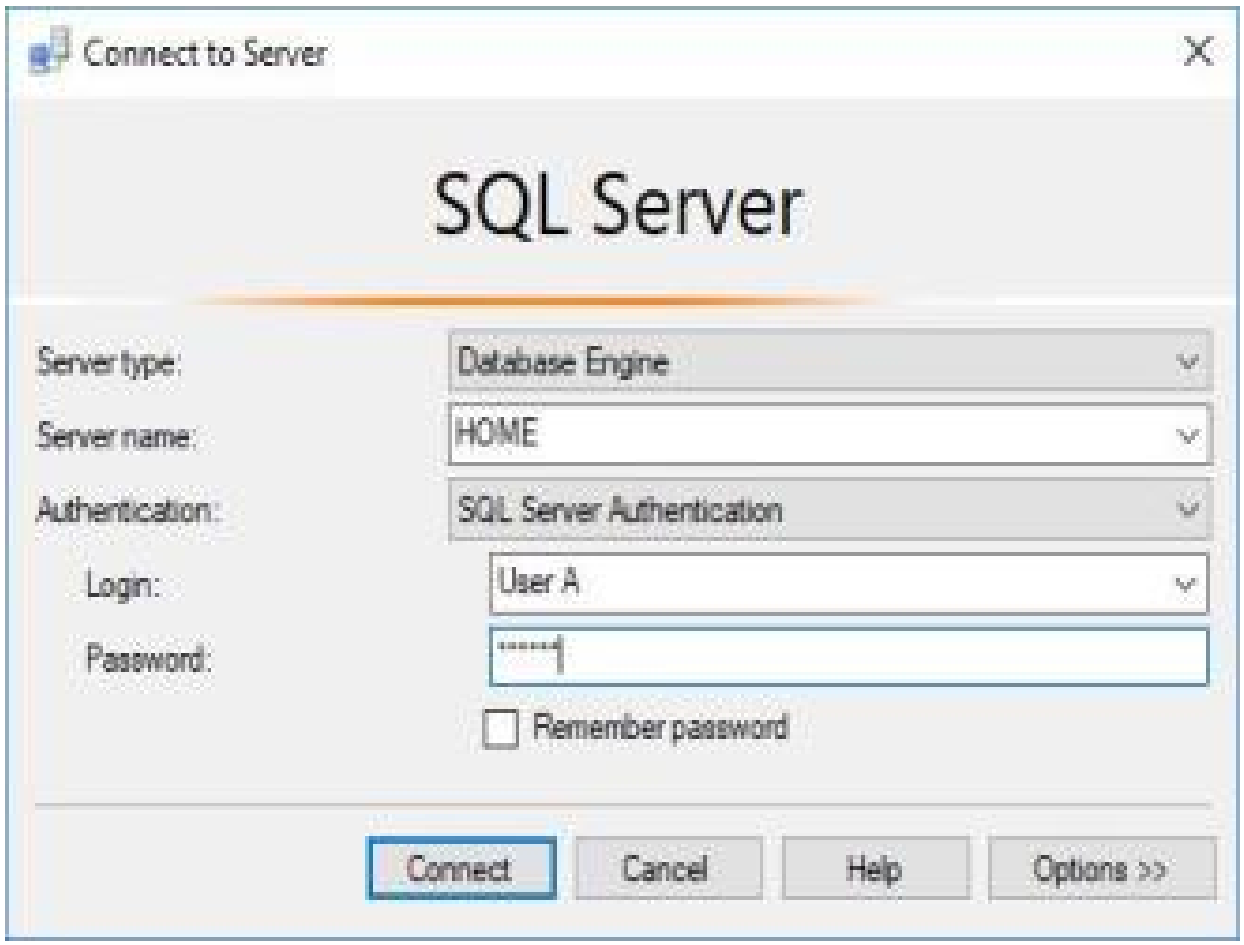
For the above user that you created, called Server User, give it a server role of serveradmin.

After you’ve successfully completed the exercise, you’ll receive a message telling you that your command has been completed successfully. You can now login to SQL Server using that account!

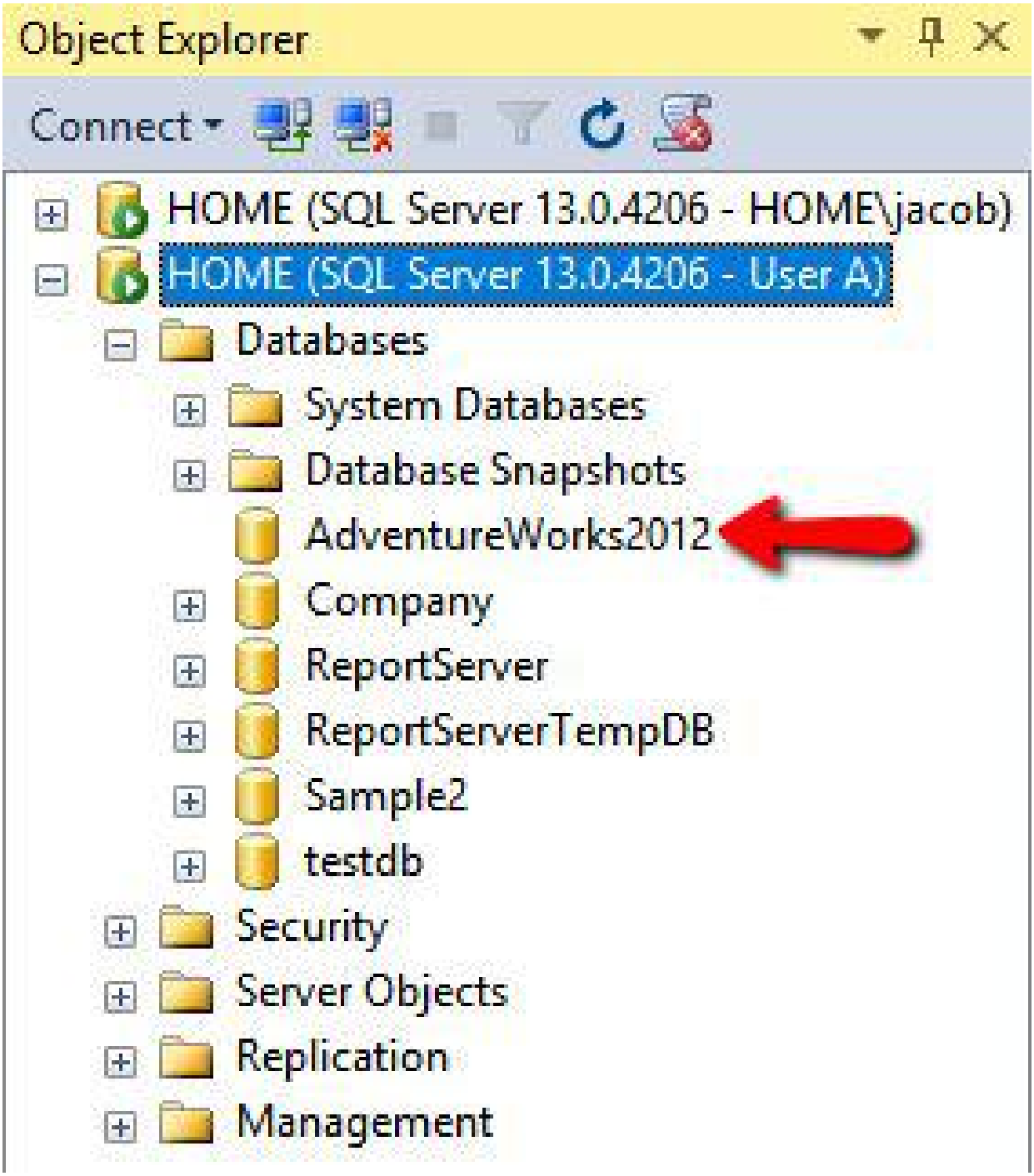
To log in, just navigate to the ‘Connect Object Explorer’ button in the Object Explorer window and click it to bring up the connection window.



When the window comes up to login, select the drop-down where it says 'Windows Authentication' and select 'SQL Server Authentication' instead. Just enter the user name and password and log in!



Once you are able to log in with this user, try to open the databases (other than the system databases) in the Object Explorer. It should look something like this; when you try to expand a database, it displays as blank instead.



You shouldn't be able to since this user doesn't have any database access. We'll cover this more thoroughly in the next section.

Database Users

A database user is a user that's typically associated with a login and mapped to a database or set of databases. The purpose of this

database user is to provide access to the databases within the server itself. You can also restrict access to other databases.

In many cases, there are multiple databases on one server instance and you wouldn't want to give all database access to all of the users.

Database Level Roles

Much like the server level roles, there are database level roles that can be assigned to a user to control their permissions within certain databases.

There are nine predefined database roles:

Db_owner - provides the ability to perform all configuration and maintenance jobs on the database. This role also allows a user to DROP databases - a powerful command!

Db_securityadmin - provides the ability to modify role membership and manage permissions.

Db_accessadmin – provides the ability to add or remove access to the database for Windows logins, Windows groups and SQL logins.

Db_backupoperator - allows the user to back up the database.

Db_ddladmin - provides the ability to run any DDL statement in SQL Server.

Db_datawriter - provides the ability to add, delete or modify data within any user table.

Db_datareader - provides the ability to view any data from user tables.

Db_denydatawriter - cannot add, delete or modify data within any user table.

db_denydatareader - cannot view any data from user tables.

Assigning Database Roles and Creating Users

Like assigning server roles and creating logins, you can use either Management Studio or SQL syntax to create a database user and assign database level roles to that particular user. You can also map that user to a login so that the two are correlated.

Below is some sample syntax to associate a database user to a login, as well as assigning it a database role. It can become a little extensive, but I've broken it down into four parts to make it easier.

```
--Using the database that we'd like to add the user to
USE [AdventureWorks2012]
GO
--Creating the user called 'User A' for the login 'User A'
CREATE USER [User A] FOR LOGIN [User A]
GO
--Using AdventureWorks2012 again, since the database level role needs to be properly
--mapped to the user
USE [AdventureWorks2012]
GO
--Altering the role for db_datawriter and adding the database user 'User A' so
--the user can add, delete or modify existing data within AdventureWorks2012
ALTER ROLE [db_datawriter] ADD MEMBER [User A]
GO
```

LIKE Clause

SQL provides us with the LIKE clause that helps us compare a value to similar values using the wildcard operators.

The wildcard operators that are used together with the LIKE clause include the percentage sign (%) and the underscore (_).

The symbols have the syntax below:

```
SELECT FROM tableName
WHERE column LIKE 'XXX%'
```

Or the following syntax:

```
SELECT FROM tableName
WHERE column LIKE '%XXX%'
```

Or the following syntax:

```
SELECT FROM tableName
WHERE column LIKE 'XXX_'
```

Or the following syntax:

```
SELECT FROM tableName
WHERE column LIKE '_XXX'
```

Or the following syntax:

```
SELECT FROM tableName
WHERE column LIKE '_XXX_'
```

If you have multiple conditions, combine them using the conjunctive operators (AND, OR). The XXX in the above case represents any string or numerical value.

Let’s describe the meaning of some of the statements that you can create with the LIKE clause:

1. WHERE SALARY LIKE '300%'

To return any values that begin with 300.

2. WHERE SALARY LIKE '%300%'

This will return any values with 300 anywhere.

3. WHERE SALARY LIKE '_00%'

This will find the values with 00 in second and third positions.

4. WHERE SALARY LIKE '4_%_%'

This will find the values that begin with 3 and that are at least 3 characters long.

5. WHERE SALARY LIKE '%3'

This will find the values that end with a 3.

6. WHERE SALARY LIKE '_3%4'

This will look for the values with a 3 in the second position and end with a 4.

7. WHERE SALARY LIKE '3___4'

This will find the values in a 5 digit number tha begin with a 3 and end with a 4.

Now, we need to demonstrate how to use this clause. We will use the EMPLOYEES table with the data shown below:[42]

ID	NAME	ADDRESS	AGE	SALARY
2	Mercy	Mercy32	25	3500.00
3	Joel	Joel42	30	4000.00
4	Alice	Alice442	31	2500.00
5	Nicholas	nicoh442	45	5000.00
6	Milly	mil342	32	2000.00
7	Grace	gra361	35	4000.00

Let's run a command that shows us all records in which the value of salary begins with 400:

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

The command returns the following:

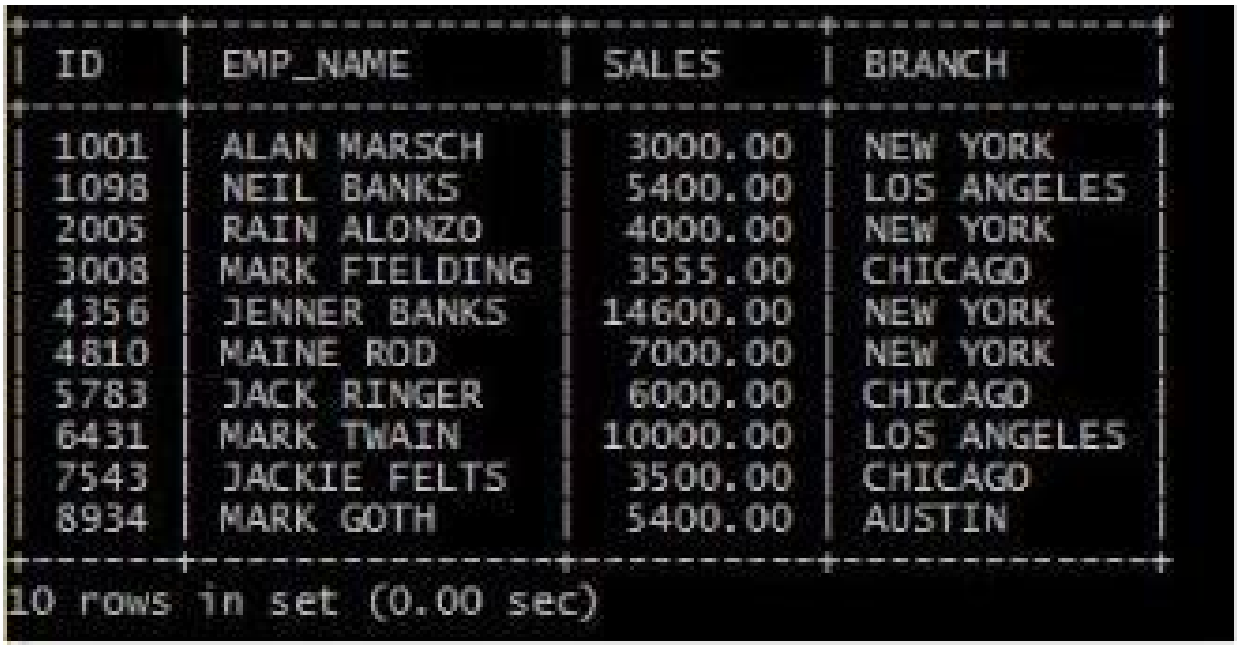
```
mysql> SELECT * FROM EMPLOYEES
    -> WHERE SALARY LIKE '400%';
+----+-------+---------+-----+---------+
| ID | NAME  | ADDRESS | AGE | SALARY  |
+----+-------+---------+-----+---------+
|  3 | Joel  | Joel42  |  30 | 4000.00 |
|  7 | Grace | gra361  |  35 | 4000.00 |
+----+-------+---------+-----+---------+
2 rows in set (0.13 sec)

mysql>
```

SQL Functions[43]

Here's the syntax:

SELECT COUNT (<expression>)

FROM table_name:



ID	EMP_NAME	SALES	BRANCH
1001	ALAN MARSCH	3000.00	NEW YORK
1098	NEIL BANKS	5400.00	LOS ANGELES
2005	RAIN ALONZO	4000.00	NEW YORK
3008	MARK FIELDING	3555.00	CHICAGO
4356	JENNER BANKS	14600.00	NEW YORK
4810	MAINE ROD	7000.00	NEW YORK
5783	JACK RINGER	6000.00	CHICAGO
6431	MARK TWAIN	10000.00	LOS ANGELES
7543	JACKIE FELTS	3500.00	CHICAGO
8934	MARK GOTH	5400.00	AUSTIN

10 rows in set (0.00 sec)

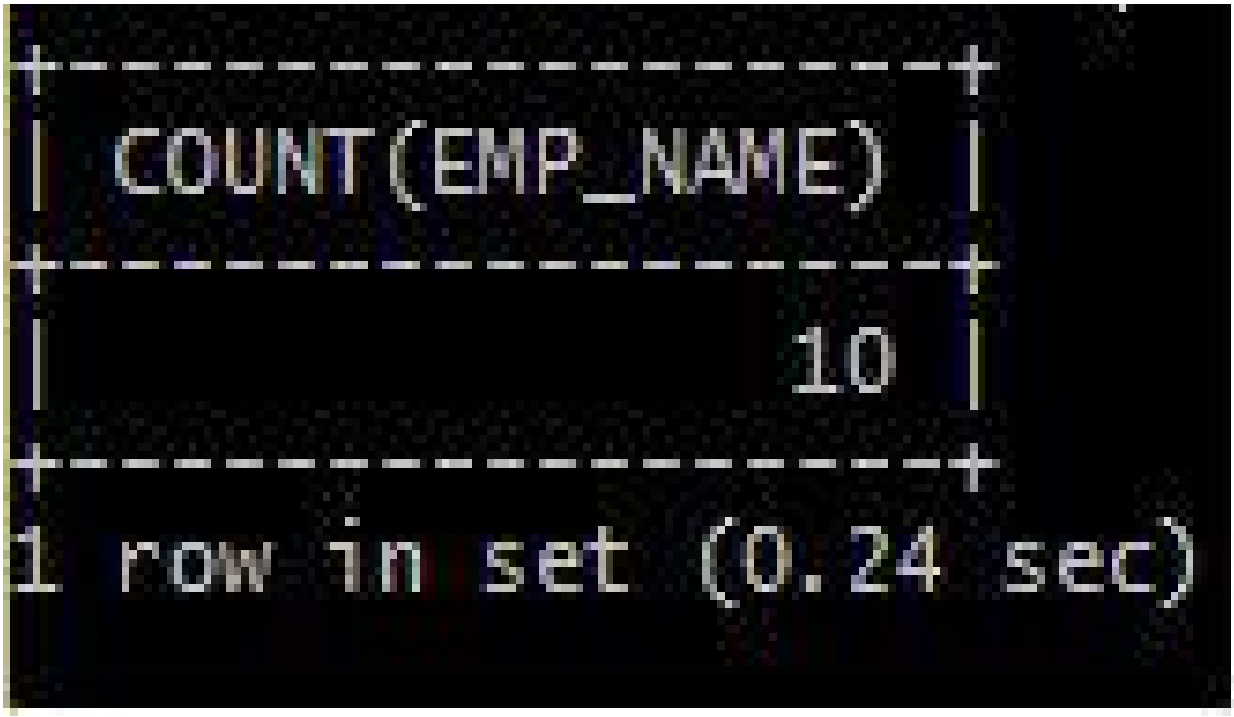
In this statement, the expression can refer to an arithmetic operation or column name. You can also specify (*) if you want to calculate the total records stored in the table.[44]

To perform a simple count operation, like calculating how many rows are in the SALES_REP table, enter:

SELECT COUNT(EMP_NAME)

FROM SALES_REP;

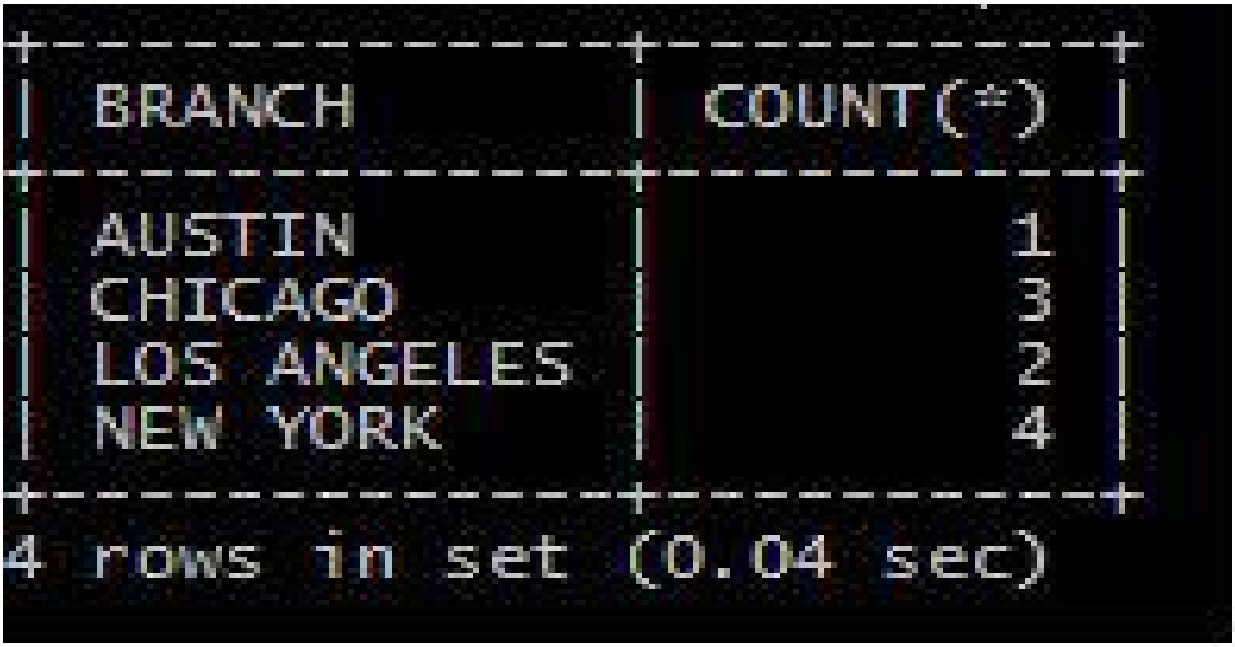
Here’s the result:



You can also use (*) instead of specifying a column name:[45]

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

This statement will produce the same result because the EMP_NAME field has no NULL value. Assuming, however, that one of the fields in the EMP_NAME contains a NULL value, this would not be included in the statement that specifies EMP_NAME but will be included in the COUNT() result if you use the * symbol as a parameter.



BRANCH	COUNT(*)
AUSTIN	1
CHICAGO	3
LOS ANGELES	2
NEW YORK	4

4 rows in set (0.04 sec)

SELECT BRANCH, COUNT(*) FROM SALES_REP

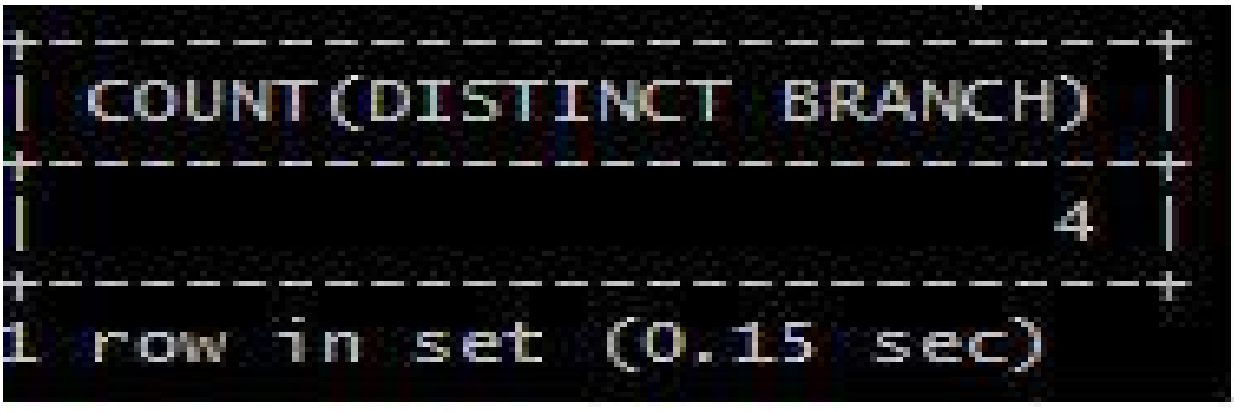
GROUP BY BRANCH;

This would be the output:

The COUNT() function can be used with DISTINCT to find the number of distinct entries. For instance, if you want to know how many distinct branches are saved in the SALES_REP table, enter the following statement:

SELECT COUNT (DISTINCT BRANCH)

FROM SALES_REP;



COUNT(DISTINCT BRANCH)
4

1 row in set (0.15 sec)

SQL AVG Function[46]

Here is the syntax:

SELECT AVG (<expression>)

FROM “table_name”;

ID	EMP_NAME	SALES	BRANCH
1001	ALAN MARSCH	3000.00	NEW YORK
1098	NEIL BANKS	5400.00	LOS ANGELES
2005	RAIN ALONZO	4000.00	NEW YORK
3008	MARK FIELDING	3555.00	CHICAGO
4356	JENNER BANKS	14600.00	NEW YORK
4810	MAINE ROD	7000.00	NEW YORK
5783	JACK RINGER	6000.00	CHICAGO
6431	MARK TWAIN	10000.00	LOS ANGELES
7543	JACKIE FELTS	3500.00	CHICAGO
8934	MARK GOTH	5400.00	AUSTIN

10 rows in set (0.00 sec)

In the above statement, the expression can refer to an arithmetic operation or to a column name. Arithmetic operations can have single or multiple columns.

In the first example, you will use the AVG() function to calculate the average sales amount. You can enter the following statement:[47]

AVG(SALES)
6245.500000

1 row in set (0.00 sec)

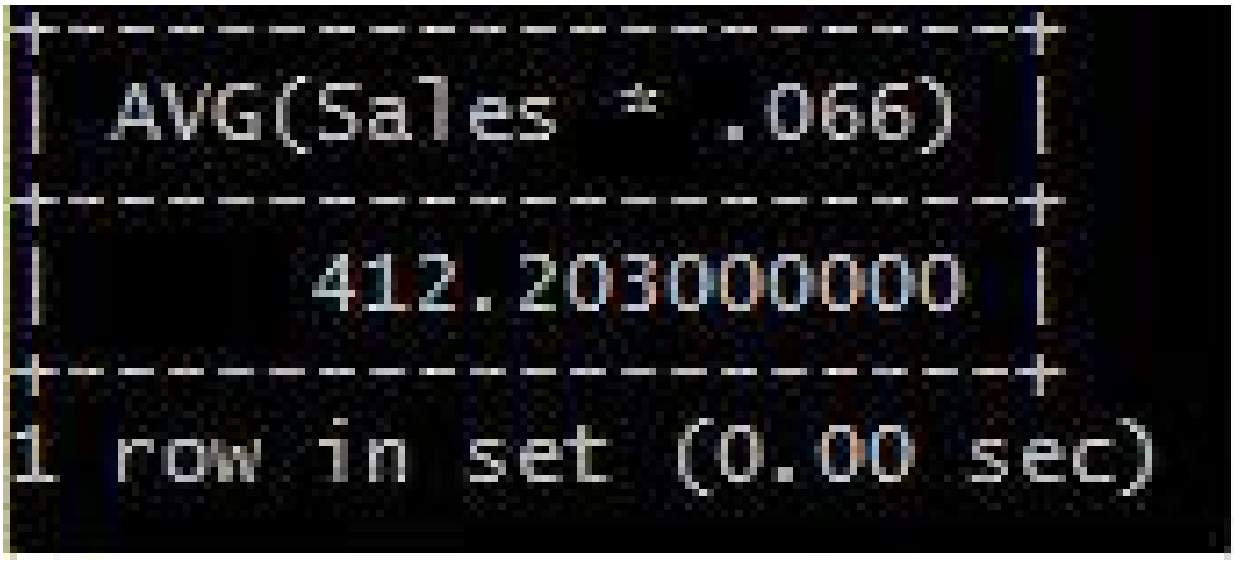
SELECT AVG(Sales) FROM Sales_Rep;

This is the result:

https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017

The figure 6245.500000 is the average of all sales data in the Sales_Rep table and it is computed by adding the Sales field and dividing the result by the number of records which, in this example, is 10 rows.

The AVG() function can be used in arithmetic operations. For example, assuming that sales tax is 6.6% of sales, you can use this statement to calculate the average sales tax figure:



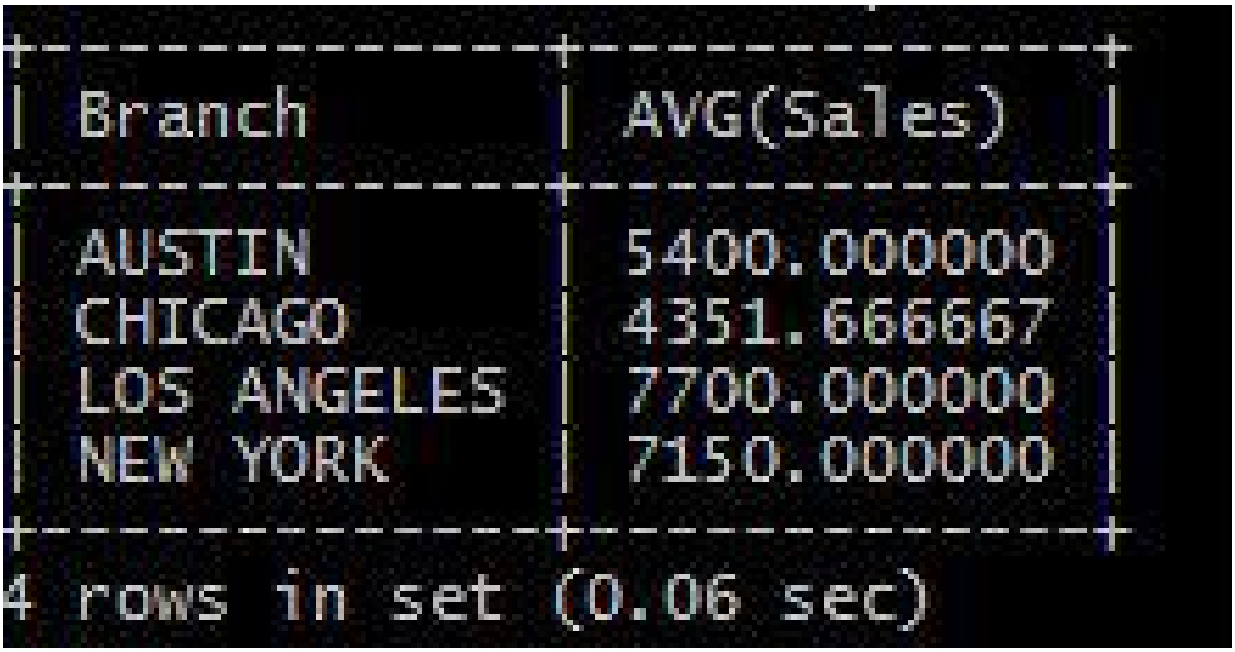
SELECT AVG(Sales*.066) FROM Sales_Rep;

Here's the result:

https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017

To obtain the result, SQL had to calculate the result of the arithmetic operation 'Sales *.o66' before applying the AVG function.

SELECT Branch, AVG(Sales) FROM Sales_Rep



Branch	AVG(Sales)
AUSTIN	5400.000000
CHICAGO	4351.666667
LOS ANGELES	7700.000000
NEW YORK	7150.000000

4 rows in set (0.06 sec)

GROUP BY Branch;

Here's the result:

https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017

SQL ROUND Function

The following is the syntax for SQL ROUND() function:

ROUND (expression, [decimal place])

In the above statement, the decimal place specifies the number of decimal points that will be returned. For instance, specifying -1 will round off the number to the nearest tens.

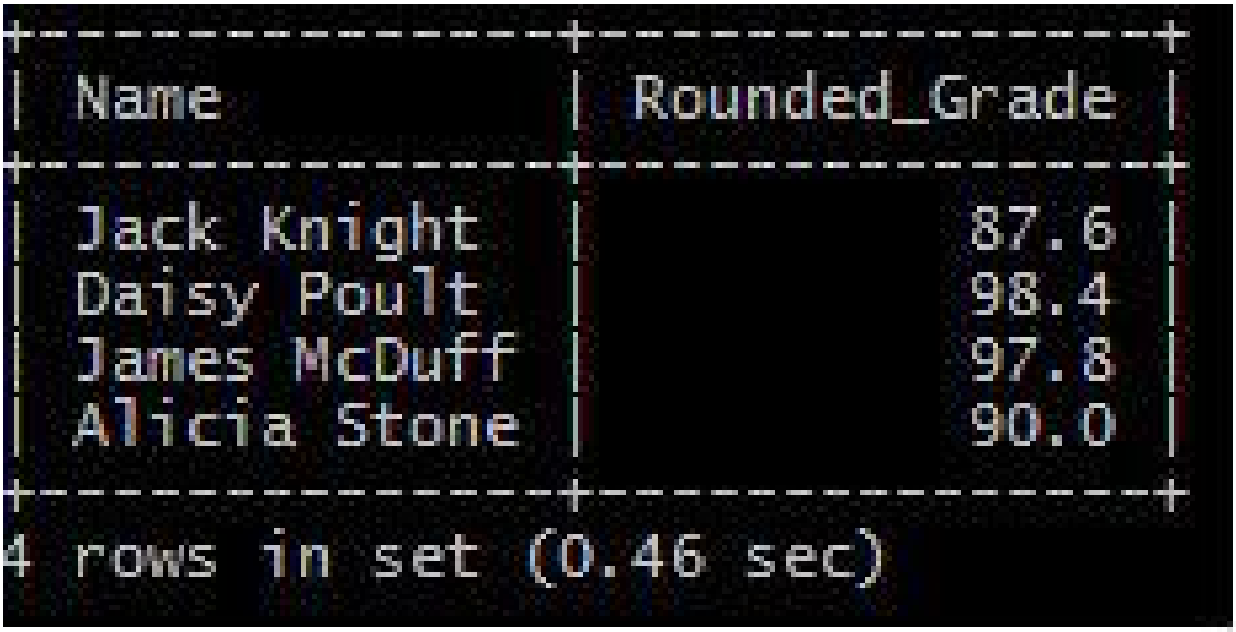
The examples on this section will use the Student_Grade table with the following data:

ID	Name	Grade
1	Jack Knight	87.6498
2	Daisy Poult	98.4359

3	James	McDuff	97.7853
4	Alicia	Stone	89.9753

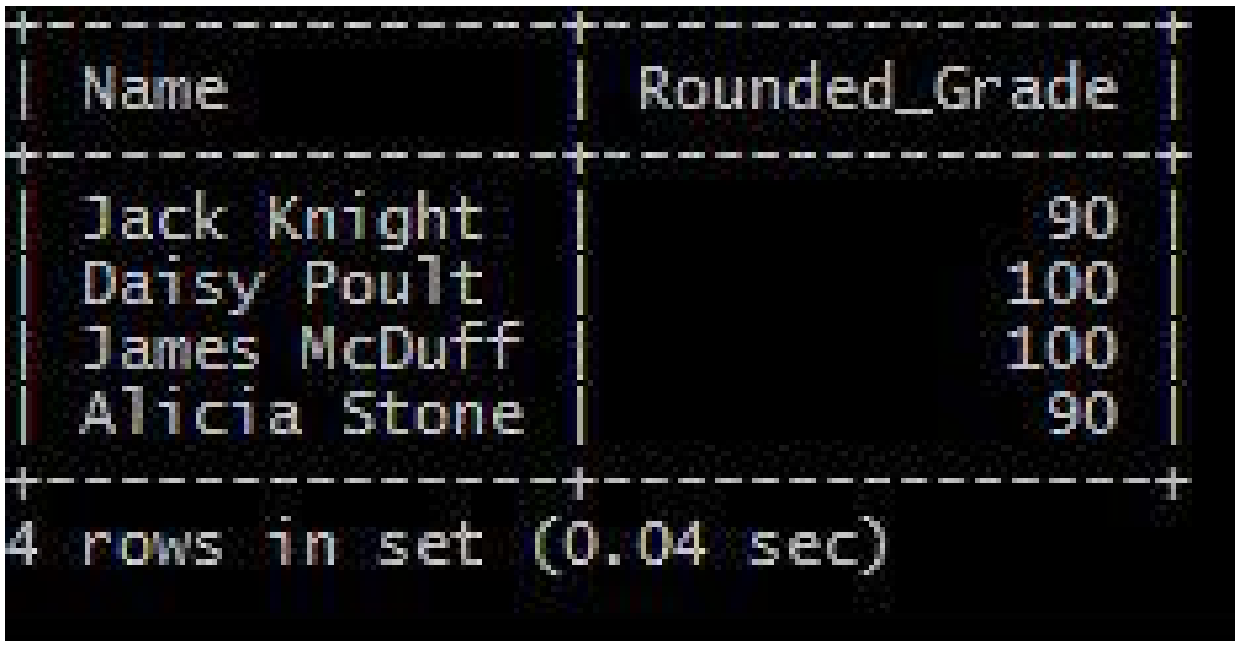
To round off the grades to the nearest tenths, enter the following statement:

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



This would be the result:[48]

Assuming that you want to round the grades to the nearest tens, you will use a negative parameter for the ROUND() function:



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

4 rows in set (0.04 sec)

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

Here's the result:

SQL SUM Function

The SUM() function is used to return the total for an expression.

Here's the syntax for the SUM() function:

SELECT SUM (<expression>)

FROM "table_name";

The expression parameter can refer to an arithmetic operation or a column name. Arithmetic operations may include one or more columns.

Likewise, there can be more than one column in the SELECT statement in addition to the column specified in the SUM() function. These columns should also form part of the GROUP BY clause. Here's the syntax:[49]

ID	EMP_NAME	SALES	BRANCH
1001	ALAN MARSCH	3000.00	NEW YORK
1098	NEIL BANKS	5400.00	LOS ANGELES
2005	RAIN ALONZO	4000.00	NEW YORK
3008	MARK FIELDING	3555.00	CHICAGO
4356	JENNER BANKS	14600.00	NEW YORK
4810	MAINE ROD	7000.00	NEW YORK
5783	JACK RINGER	6000.00	CHICAGO
6431	MARK TWAIN	10000.00	LOS ANGELES
7543	JACKIE FELTS	3500.00	CHICAGO
8934	MARK GOTH	5400.00	AUSTIN

10 rows in set (0.00 sec)

SELECT colum n1, colum n2, … column N, SUM (“colum nN+1”)

FROM table_name;

GROUP BY colum n1, colum n2, … colum n_nameN;

For the examples in this section, you will use the SALES_REP table with the following data:[50]

To calculate the total of all sales from the Sales_Rep table, enter the following statement:

SELECT SUM(Sales) FROM Sales_Rep;

SUM(Sales)
62455.00

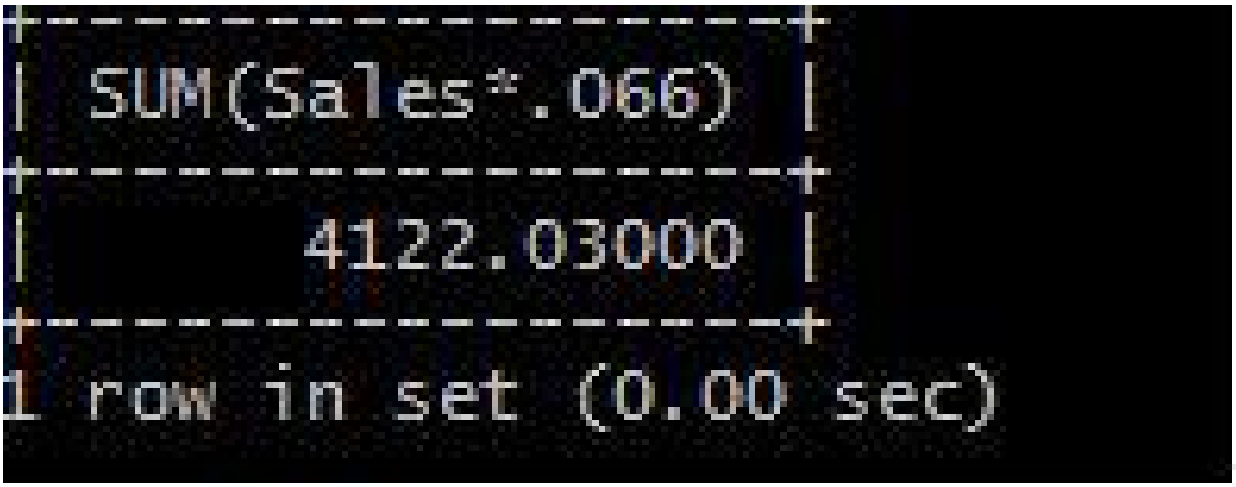
1 row in set (0.00 sec)

This would be the result:[51]

The figure 62455.00 represents the total of all entries in the Sales column.

To illustrate how you can use an arithmetic operation as an argument in the SUM() function, assume that you have to apply a sales tax of 6.6% on the sales figure. Here's the statement to obtain the total sales tax:

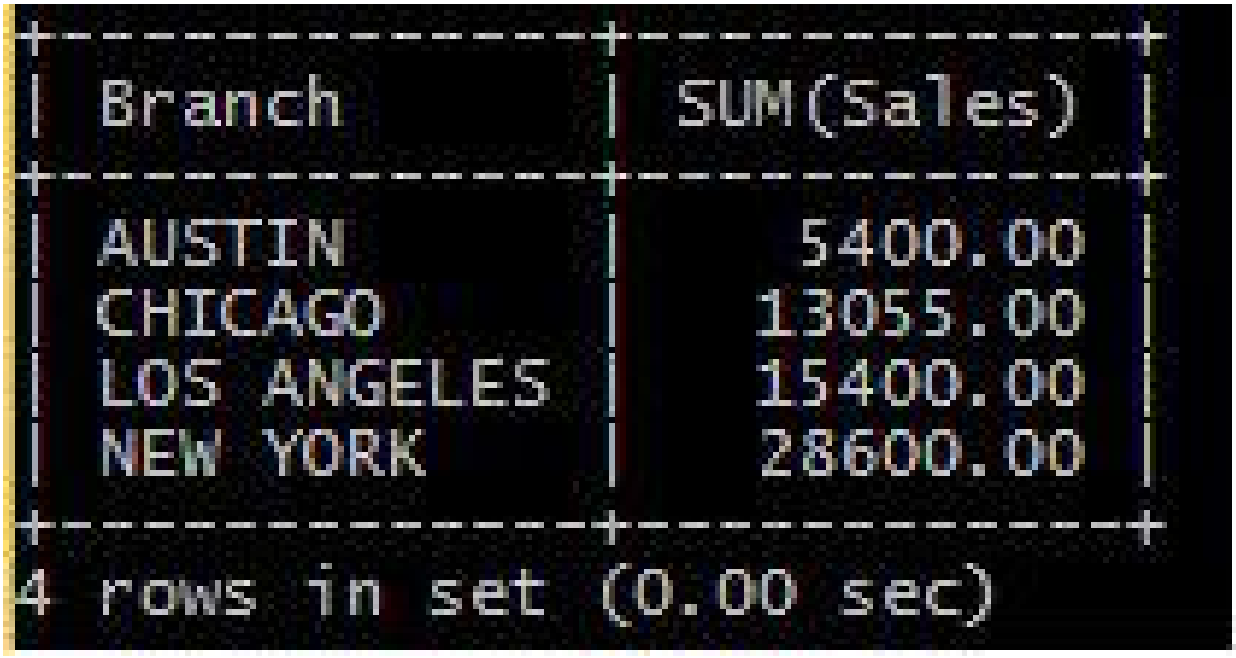
```
SELECT SUM(Sales*.066) FROM Sales_Rep;
```



You will get the following result:[52]

In this example, you will combine the SUM() function and the GROUP BY clause to calculate the total sales for each branch. You can use the following statement:

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



Branch	SUM(Sales)
AUSTIN	5400.00
CHICAGO	13055.00
LOS ANGELES	15400.00
NEW YORK	28600.00

4 rows in set (0.00 sec)

Here's the result:[53]

SQL MAX() Function

The MAX() function is used to obtain the largest value in a given expression.

Here's the syntax:

SELECT MAX (<expression>)

FROM table_name;

The expression parameter can be an arithmetic operation or a column name. Arithmetic operations can have multiple columns.

The SELECT statement can have one or more columns, aside from the column specified in the MAX() function. If this is the case, these columns will have to form part of the GROUP BY clause.

The syntax would be:

SELECT column1, column2, ... "columnN", MAX (<expression>)

FROM table_name;

GROUP BY column1, column2, ... "columnN";

ID	EMP_NAME	SALES	BRANCH
1001	ALAN MARSCH	3000.00	NEW YORK
1098	NEIL BANKS	5400.00	LOS ANGELES
2005	RAIN ALONZO	4000.00	NEW YORK
3008	MARK FIELDING	3555.00	CHICAGO
4356	JENNER BANKS	14600.00	NEW YORK
4810	MAINE ROD	7000.00	NEW YORK
5783	JACK RINGER	6000.00	CHICAGO
6431	MARK TWAIN	10000.00	LOS ANGELES
7543	JACKIE FELTS	3500.00	CHICAGO
8934	MARK GOTH	5400.00	AUSTIN

10 rows in set (0.00 sec)

To demonstrate this, you will use the Sales_Rep table with this data:[54]

To get the highest sales amount, enter the following statement:

SELECT MAX(Sales) FROM Sales_Rep;

Here's the result:

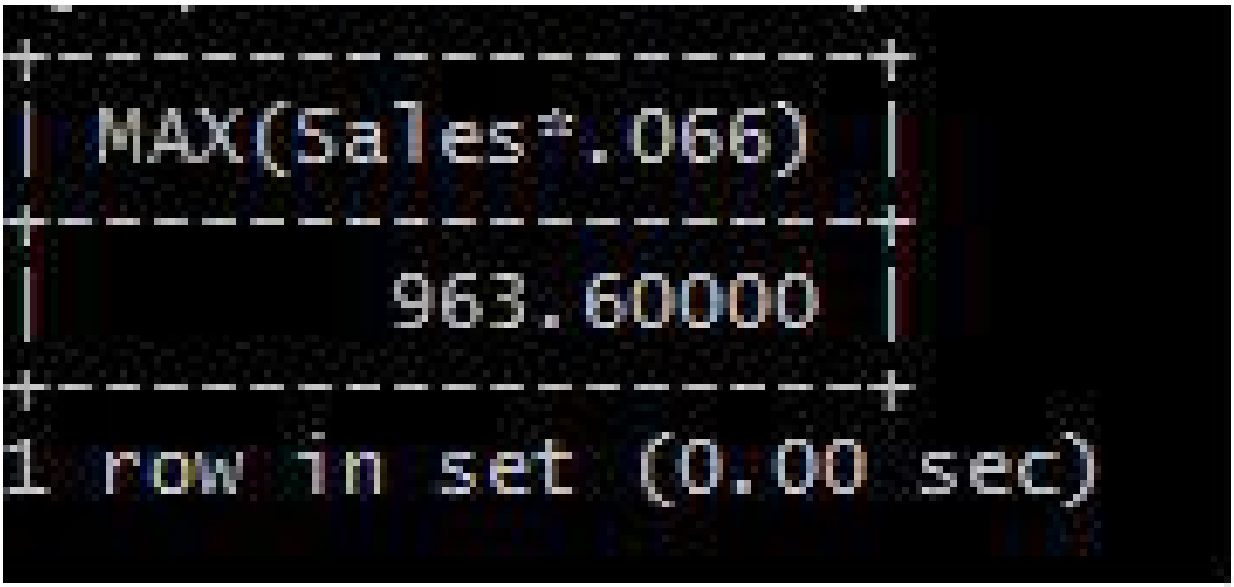
MAX(Sales)
14600.00

1 row in set (0.08 sec)

To illustrate how the MAX() function is applied to an arithmetic operation, assume that you have to compute a sales tax of 6.6% on

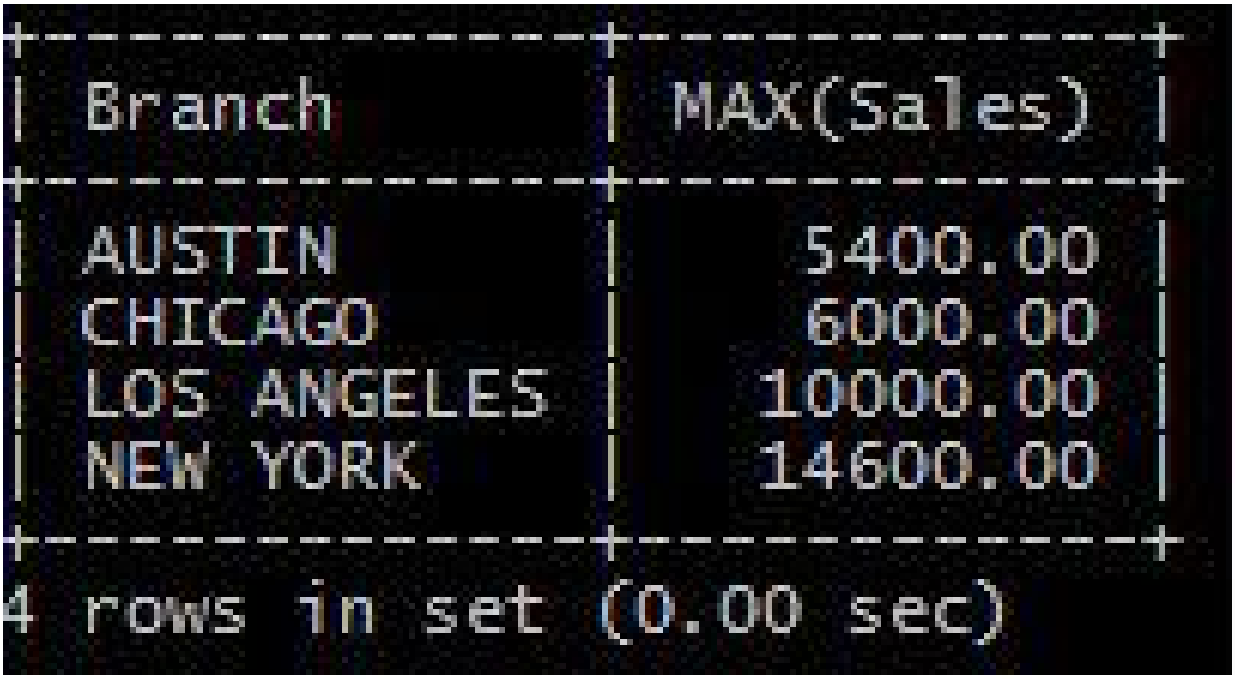
the sales figure. To get the highest sales tax figure, use the following statement;

SELECT MAX(Sales*0.066) FROM Sales_Rep;[55]



Here's the output:

You can combine the MAX() function with the GROUP BY clause to obtain the maximum sales value per branch. To do that, enter the following statement:



SELECT Branch, MAX(Sales) FROM Sales_Rep GROUP BY Branch;[56]

SQL MIN() Function

The MIN() function is used to obtain the lowest value in a given expression.

Here's the syntax:

SELECT MIN(<expression>)

FROM table_name;

The expression parameter can be an arithmetic operation or a column name. Arithmetic operations can also have several columns.

The SELECT statement can have one or several columns aside from the column specified in the MIN() function. If this is the case, these columns will have to form part of the GROUP BY clause.

The syntax would be:

SELECT column1, column2, ... "columnN", MIN (<expression>)

FROM table_name;

GROUP BY column1, column2, ... "columnN";

To demonstrate how the MIN() function is used in SQL, use the Sales_Rep table with the following data:[57]

ID	EMP_NAME	SALES	BRANCH
1001	ALAN MARSCH	3000.00	NEW YORK
1098	NEIL BANKS	5400.00	LOS ANGELES
2005	RAIN ALONZO	4000.00	NEW YORK
3008	MARK FIELDING	3555.00	CHICAGO
4356	JENNER BANKS	14600.00	NEW YORK
4810	MAINE ROD	7000.00	NEW YORK
5783	JACK RINGER	6000.00	CHICAGO
6431	MARK TWAIN	10000.00	LOS ANGELES
7543	JACKIE FELTS	3500.00	CHICAGO
8934	MARK GOTH	5400.00	AUSTIN

10 rows in set (0.00 sec)

To get the lowest sales amount, use the following statement:

SELECT MIN(Sales) FROM Sales_Rep;

The output would be: [58]

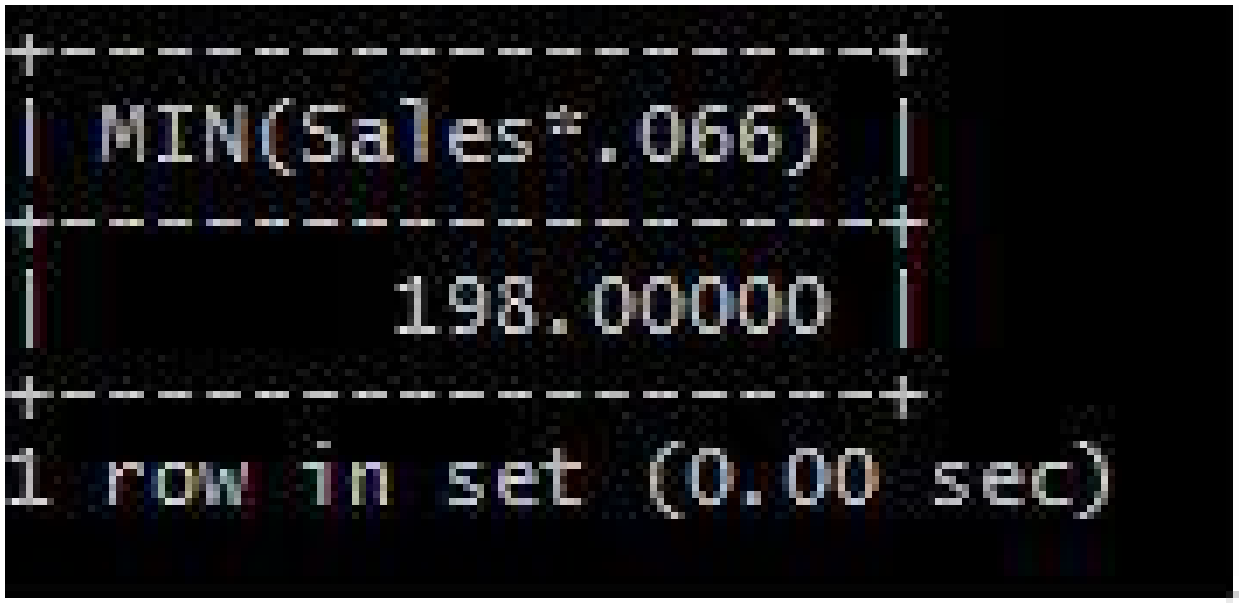
MIN(Sales)
3000.00

1 row in set (0.01 sec)

To demonstrate how the MIN() function is used on arithmetic operations, assume that you have to compute a sales tax of 6.6% on the sales figure. To get the lowest sales tax figure, use the following statement:

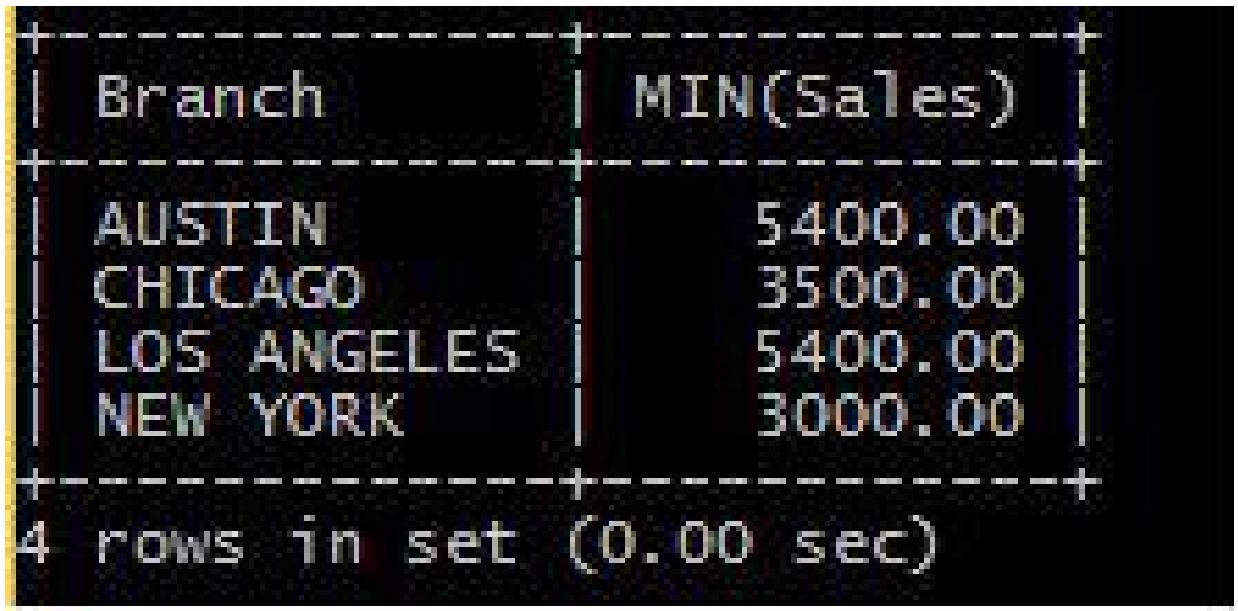
SELECT MIN(Sales*0.066) FROM Sales_Rep;

Here's the output:



You can also use the MIN() function with the GROUP BY clause to calculate the minimum sales value per branch. To do so, enter the following statement:

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



Here's the result:[59]

CHAPTER 9: Modifying and Controlling Tables

“ The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.” – Theodore Rubin

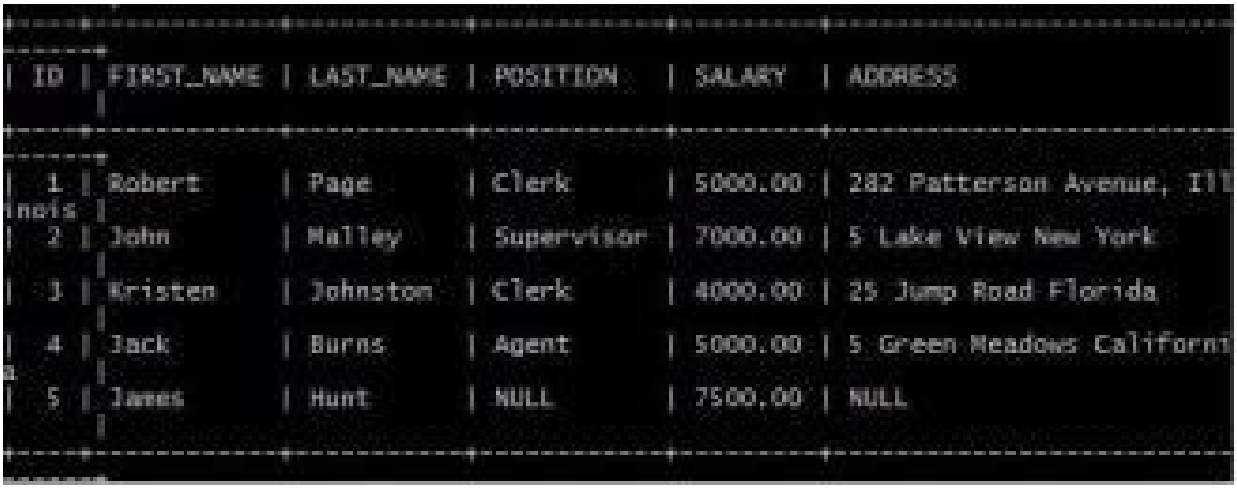
Here’s the basic syntax to alter a table:

```
ALTER TABLE TABLE_NAME [MODIFY] [COLUMN
 COLUMN_NAME][DATATYPE | NULL NOT NULL]
[RESTRICT | CASCADE]
    [DROP] [CONSTRAINT CONSTRAINT_NAME]
    [ADD] [COLUMN] COLUMN DEFINITION
```

Changing a Table’s Name

The ALTER TABLE command can be used with the RENAME function to change a table’s name.

To demonstrate the use of this statement, use the EMPLOYEES table with the following records:



ID	FIRST_NAME	LAST_NAME	POSITION	SALARY	ADDRESS
1	Robert	Page	Clerk	5000.00	282 Patterson Avenue, Illinois
2	John	Malley	Supervisor	7000.00	5 Lake View New York
3	Kristen	Johnston	Clerk	4000.00	25 Jump Road Florida
4	Jack	Burns	Agent	5000.00	5 Green Meadows California
5	James	Hunt	NULL	7500.00	NULL

To change the EMPLOYEES table name to INVESTORS, use the following statement:[60]

```
ALTER TABLE EMPLOYEES RENAME INVESTORS;
```

Your table is now called INVESTORS.

Modifying Column Attributes

A column's attributes refer to the properties and behaviors of data entered in a column. Normally, you set the column attributes when you create the table. However, you may still change one or more attributes using the ALTER TABLE command.

You may modify the following:

- Column name
- Column Data type assigned to a column
- The scale, length, or precision of a column
- Use or non-use of NULL values in a column

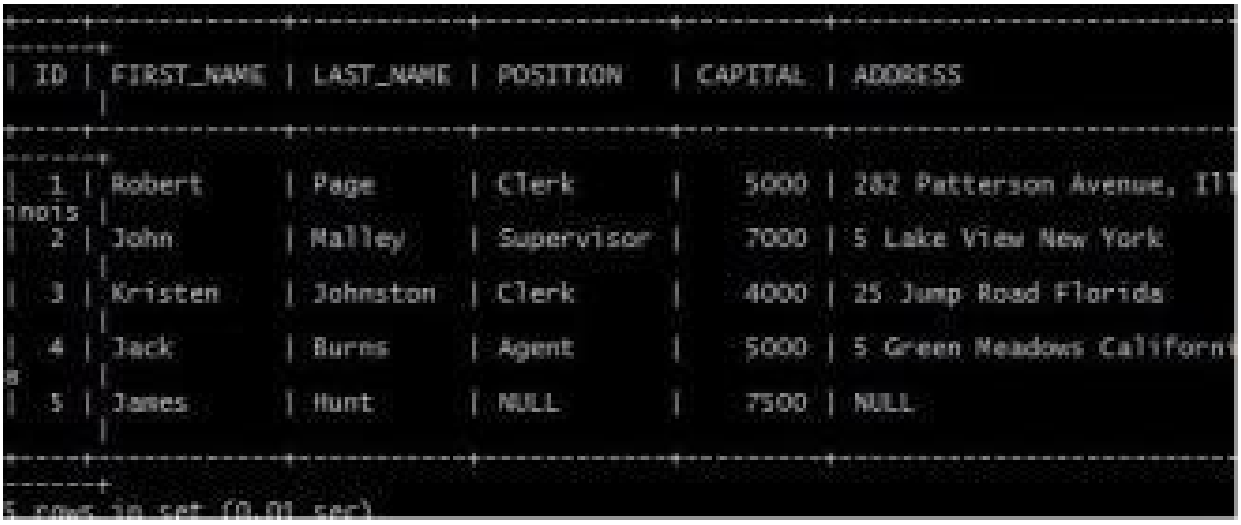
Renaming Columns

You may want to modify a column's name to reflect the data it contains. For instance, since you renamed the EMPLOYEES database to INVESTORS, the SALARY column will no longer be appropriate. You can change the column name to something like CAPITAL. Likewise, you may want to change its data type from DECIMAL to an INTEGER TYPE with a maximum of ten digits.

To do so, enter the following statement:

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

The result is the following:



ID	FIRST_NAME	LAST_NAME	POSITION	CAPITAL	ADDRESS
1	Robert	Page	Clerk	5000	282 Patterson Avenue, Illinois
2	John	Malley	Supervisor	7000	5 Lake View New York
3	Kristen	Johnston	Clerk	4000	25 Jump Road Florida
4	Jack	Burns	Agent	5000	5 Green Meadows California
5	James	Hunt	NULL	7500	NULL

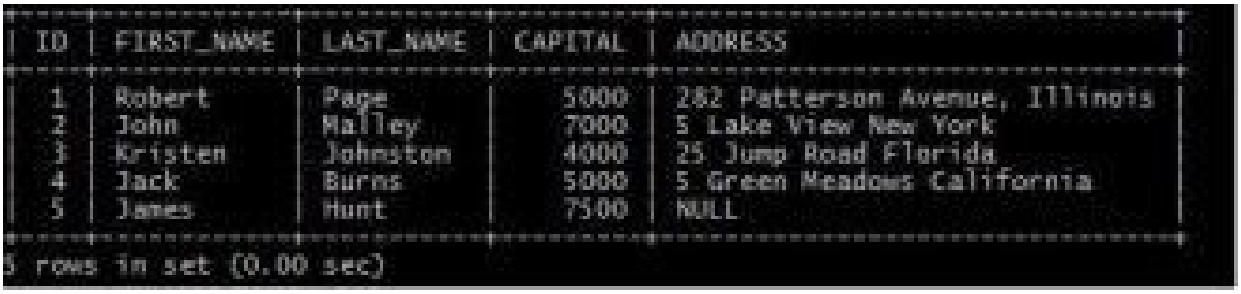
5 rows in set (0.01 sec)

Deleting a Column

At this point, the Position column is no longer applicable. You can drop the column using the following statement:

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

Here's the updated INVESTORS table: [62]



ID	FIRST_NAME	LAST_NAME	CAPITAL	ADDRESS
1	Robert	Page	5000	282 Patterson Avenue, Illinois
2	John	Malley	7000	5 Lake View New York
3	Kristen	Johnston	4000	25 Jump Road Florida
4	Jack	Burns	5000	5 Green Meadows California
5	James	Hunt	7500	NULL

5 rows in set (0.00 sec)

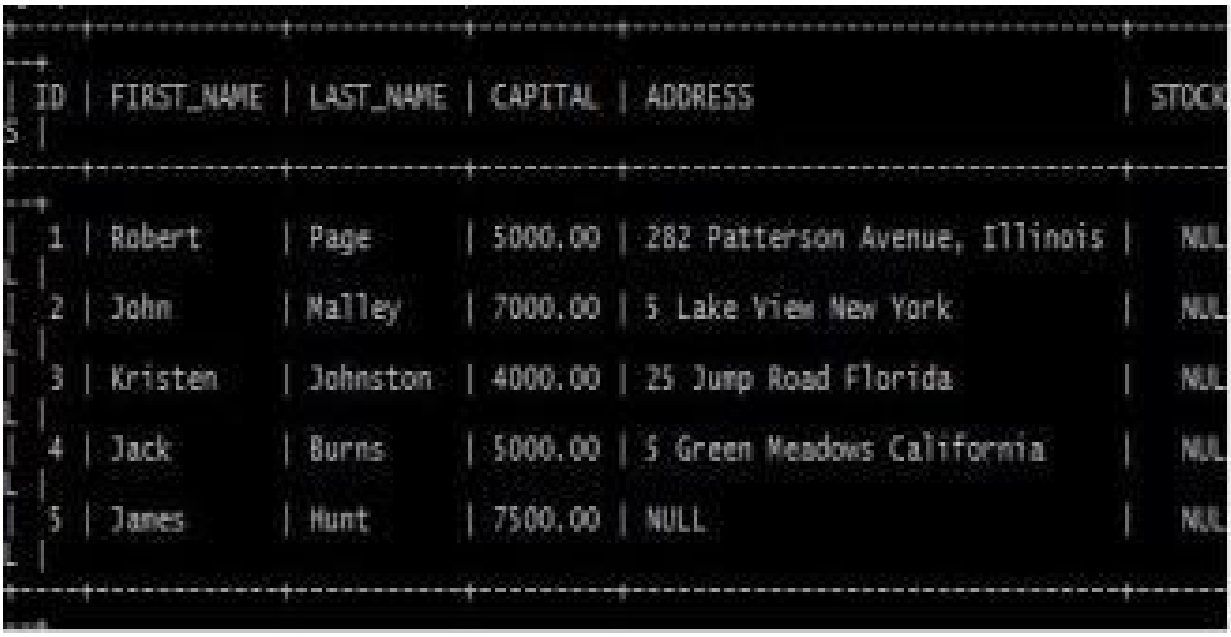
Adding a New Column

Since you're now working on a different set of data, you may decide to add another column to make the data on the INVESTORS table more relevant. You can add a column that will store the number of stocks owned by each investor. You may name the new column STOCKS. This column will accept integers up to 9 digits.

You can use the following statement to add the STOCKS column:

ALTER TABLE INVESTORS ADD STOCKS INT(9);

The following is the updated INVESTORS table: [63]



ID	FIRST_NAME	LAST_NAME	CAPITAL	ADDRESS	STOCKS
1	Robert	Page	5000.00	282 Patterson Avenue, Illinois	NULL
2	John	Malley	7000.00	5 Lake View New York	NULL
3	Kristen	Johnston	4000.00	25 Jump Road Florida	NULL
4	Jack	Burns	5000.00	5 Green Meadows California	NULL
5	James	Hunt	7500.00	NULL	NULL

Modifying an Existing Column without Changing its Name

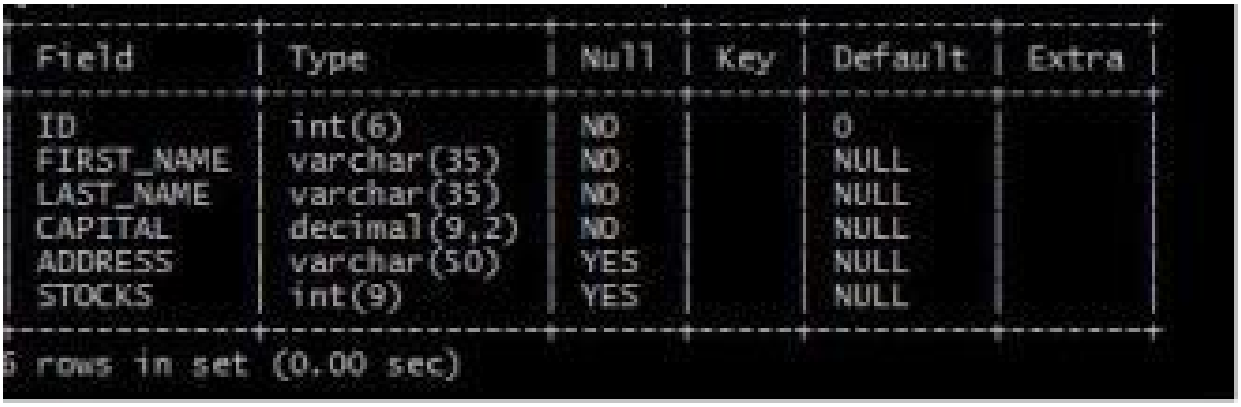
You may also combine the ALTER TABLE command with the MODIFY keyword to change the data type and specifications of a table. To demonstrate this, you can use the following statement to modify the data type of the column CAPITAL from an INT type to a DECIMAL type with up to 9 digits and two decimal numbers.

ALTER TABLE INVESTORS MODIFY CAPITAL DECIM AL(9.2) NOT NULL;

By this time, you may be curious to see the column names and attributes of the INVESTORS table. You can use the ‘SHOW COLUMNS’ statement to display the table’s structure. Enter the following statement:

SHOW COLUMNS FROM INVESTORS;

Here’s a screenshot of the result: [64]



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(50)	YES		NULL	
STOCKS	int(9)	YES		NULL	

6 rows in set (0.00 sec)

Rules to Remember when Using ALTER TABLE

- Adding Columns to a Database Table

When adding a new column, bear in mind that you can't add a column with a NOT NULL attribute to a table with existing data. You will generally specify a column to be NOT NULL to indicate that it will hold a value. Adding a NOT NULL column will contradict the constraint if the existing data don't have values for a new column.

- Modifying Fields/Columns

1. You can easily modify the data type of a column
2. .You can increase the number of digits that numeric data types hold but you will only be able to decrease it if the largest number of digits stored by a table is equal to or lower than the desired number of digits.
3. You can increase or decrease the decimal places of numeric data types as long as they don't exceed the maximum allowable decimal places.

If not handled properly, deleting and modifying tables can result to loss of valuable information. So, be extremely careful when you're executing the ALTER TABLE and DROP TABLE statements.

Deleting Tables

Dropping a table will also remove its data, associated index, triggers, constraints, and permission data. You should be careful when using this statement.

Here's the syntax:

DROP TABLE table_name;

For example, if you want to delete the INVESTORS TABLE from the xyzcompany database, you may use the following statement:

DROP TABLE INVESTORS;

The DROP TABLE command effectively removed the INVESTORS table from the current database.

If you try to access the INVESTORS table with the following command:

SELECT* FROM INVESTORS;

SQL will return an error, like this:

https://docs.microsoft.com/en-us/sql/relational-databases/tables/delete-tables-database-engine?view=sql-server-2017

Combining and joining tables

You can combine data from several tables if a common field exists between them. To do this, use the JOIN statement.

SQL supports several types of JOIN operations:

INNER JOIN

The INNER JOIN, or simply JOIN, is the most commonly used type of JOIN. It displays the rows when the tables to be joined have a matching field.

Here's the syntax: [65]

```
SELECT colum n_name(s)
FROM table1
INNER JOIN table2
ON table1.colum n_name=table2.colum n_name;
```

In this variation, the JOIN clause is used instead of INNER JOIN.

```
SELECT colum n_names(s)
FROM table1
JOIN table2
ON table1.colum n_name=table2.colum n_name;
```

LEFT JOIN

The LEFT JOIN operation returns all left table rows with the matching right table rows. If no match is found, the right side returns NULL.

[66]Here's the syntax for LEFT JOIN:

```
SELECT colum n_name(s)
FROM table1
LEFT JOIN table2
ON table1.colum n_name=table2.colum n_name;
```

In some database systems, the keyword LEFT OUTER JOIN is used instead of LEFT JOIN. Here's the syntax for this variation:[67]

```
SELECT colum n_name(s)
FROM table1
LEFT OUTER JOIN table2
```

ON table2.colum n_name=table2.colum n_name;

RIGHT JOIN

This JOIN operation returns all right table rows with the matching left table rows.

The following is the syntax for this operation: [68]

SELECT colum n_name(s)

FROM table1

RIGHT JOIN table2

ON table2.colum n_name=table2.colum n_name;

In some database systems, the RIGHT OUTER JOIN is used instead of LEFT JOIN. Here's the syntax for this variation:

SELECT colum n_name(s)

FROM table1

RIGHT OUTER JOIN table2

ON table2.colum n_name=table2.colum n_name;

http://www.sql-join.com/sql-join-types/

FULL OUTER JOIN

This JOIN operation displays all rows when at least one table meets the condition. It combines the results from both RIGHT and LEFT join operations.

Here's the syntax:

SELECT colum n_name(s)

FROM table1

FULL OUTER JOIN table2

ON table2.colum n_name=table2.colum n_name;

To demonstrate the JOIN operation in SQL, use the Branch_Sales and Branch_Location tales:

Branch_Sales Table

Branch	Product_ID	Sales
New York	101	7500.00
Los Angeles	102	6450.00
Chicago	101	1560.00
Philadelphia	101	1980.00
Denver	102	3500.00
Seattle	101	2500.00
Detroit	102	1450.00

Location Table

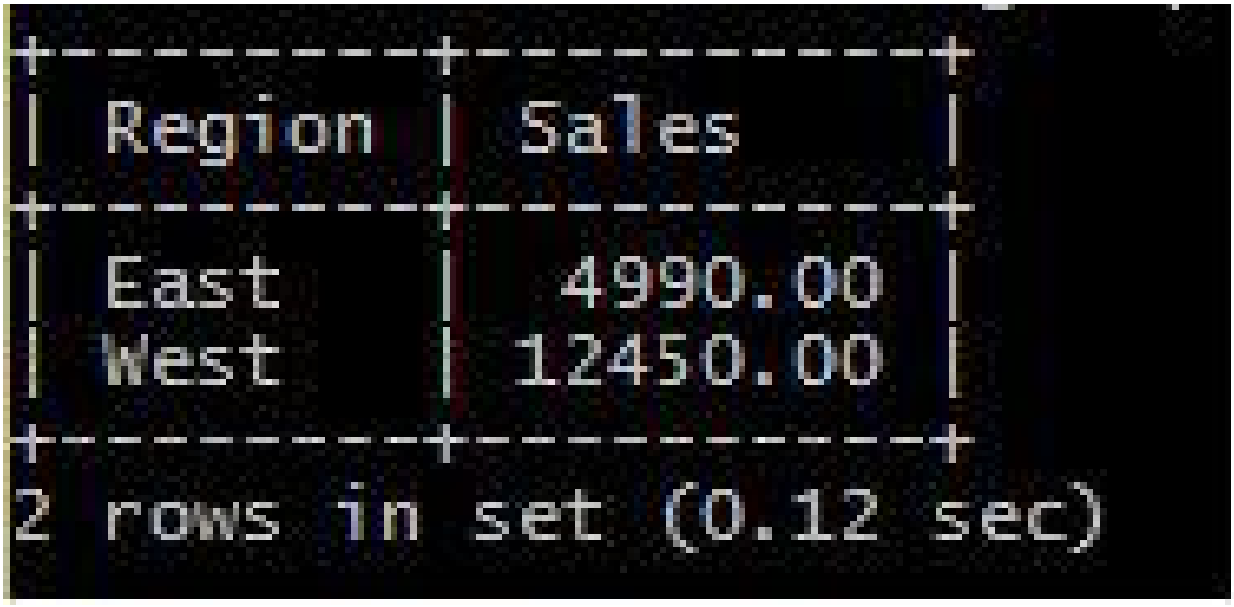
Region	Branch
East	New York City
East	Chicago
East	Philadelphia
East	Detroit
West	Los Angeles
West	Denver
West	Seattle

The objective is to fetch the sales by region. The Location table contains the data on regions and branches while the Branch_Sales table holds the sales data for each branch. To find the sales per region, you need to combine the data from the Location and Branch_Sales tables. Notice that these tables have a common field, the Branch, which is the field that links the two tables.

The following statement will demonstrate how you can link these two tables by using table aliases:

```
SELECT A1.Region Region, SUM(A2.Sales) Sales
FROM Location A1, Branch_Sales A2
WHERE A1.Branch = A2.Branch
GROUP BY A1.Region;
```

This would be the result: [69]



In the first two lines, the statement tells SQL to select the fields 'Region' from the Location table and the total of the 'Sales' field from the Branch_Sales table. The statement uses table aliases. The 'Region' field was aliased as Region while the sum of the SALES field was aliased as SALES.

Table aliasing is the practice of using a temporary name for a table or a table column. Using aliases helps make statements more readable and concise. For example, if you opt not to use a table alias for the first line, you would have used the following statement to achieve the same result:

SELECT Location.Region Region,

SUM(Branch_Sales.Sales) SALES

Alternatively, you can specify a join between two tables by using the JOIN and ON keywords. For instance, using these keywords, the query would be:

SELECT A1.Region REGION, SUM(A2.Sales) SALES

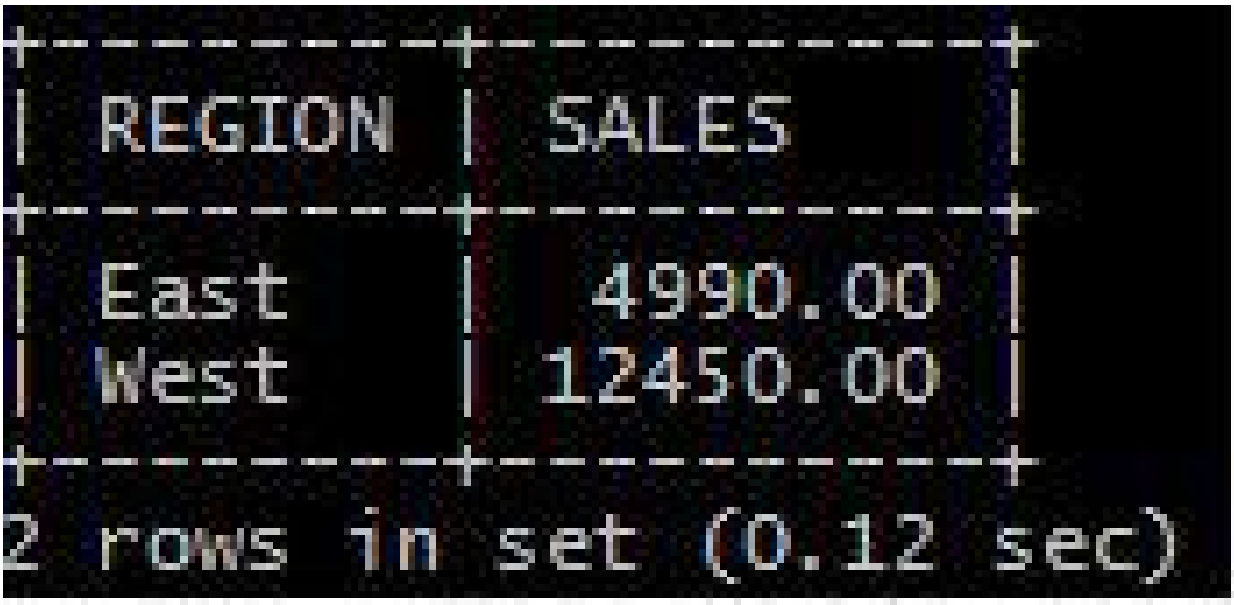
FROM Location A1

JOIN Branch_Sales A2

ON A1.Branch = A2.Branch

GROUP BY A1.Region;

The query would produce an identical result: [70]



Using Inner Join

An inner join displays rows when there is one or more matches on two tables. To demonstrate this, use the following tables:

Branch_Sales table

Branch	Product_ID	Sales
New York	101	7500.00
Philadelphia	101	1980.00
Denver	102	3500.00
Seattle	101	2500.00
Detroit	102	1450.00

Location_table

Region	Branch
East	New York
East	Chicago
East	Philadelphia
East	Detroit
West	Los Angeles
West	Denver
West	Seattle

You can achieve this by using the INNER JOIN statement.

You can enter the following:

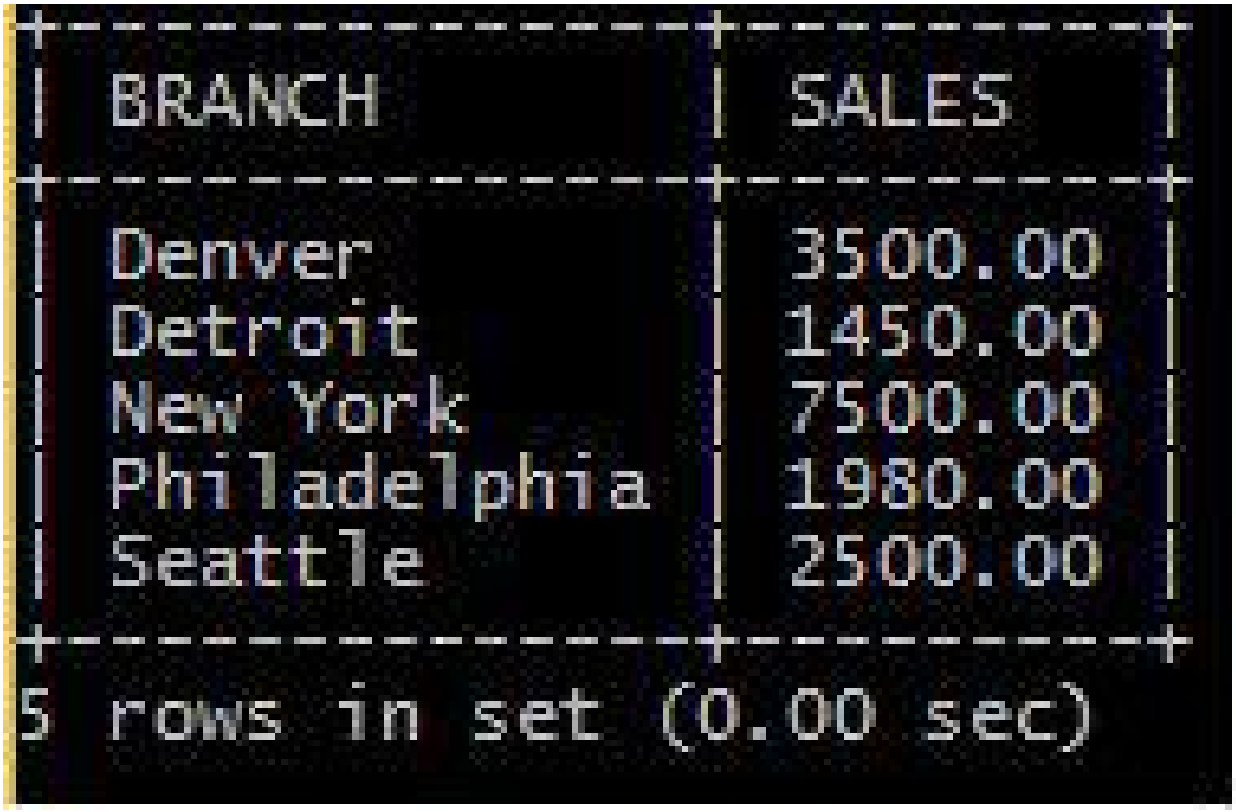
```
SELECT A1.Branch BRANCH, SUM(A2.Sales) SALES
FROM Location A1
```

INNER JOIN Branch_Sales A2

ON A1.Branch = A2.Branch

GROUP BY A1.Branch;

[71]This would be the result:



BRANCH	SALES
Denver	3500.00
Detroit	1450.00
New York	7500.00
Philadelphia	1980.00
Seattle	2500.00

5 rows in set (0.00 sec)

Take note that by using the INNER JOIN, only the branches with records in the Branch_Sales report were included in the results even though you are actually applying the SELECT statement on the Location table. The 'Chicago' and 'Los Angeles' branches were excluded because there are no records for these branches in the Branch_Sales table.

Using Outer Join

In the previous example, you have used the Inner Join to combine tables with common rows. The OUTER JOIN command is used for this purpose.

The example for the OUTER JOIN will use the same tables used for INNER JOIN: the Branch_Sales table and Location_table.

This time, you want a list of sales figures for all stores. A regular join would have excluded Chicago and Los Angeles because these branches were not part of the Branch_Sales table. Therefore, you want to do an OUTER JOIN.

The statement is the following:

```
SELECT A1.Branch, SUM(A2.Sales) SALES
FROM Location A1, Branch_Sales A2
WHERE A1.Branch = A2.Branch (+)
GROUP BY A1.Branch;
```

Please note that the Outer Join syntax is database-dependent. The above statement uses the Oracle syntax.

Here's the result: [72]

Branch	Sales
Chicago	NULL
Denver	3500.00
Detroit	1450.00
Los Angeles	NULL
New York	7500.00
Philadelphia	1980.00
Seattle	2500.00

When combining tables, be aware that some JOIN syntax have different results across database systems. To maximize this powerful database feature, it is important to read the RDBMS documentation.

LIMIT, TOP and ROWNUM Clauses

The TOP command helps us retrieve only the TOP number of records from the table. However, you must note that not all databases support the TOP command. Some will support the LIMIT while others will support the ROWNUM clause.

The following is the syntax to use the TOP command on the SELECT statement:

SELECT TOP number|percent columnName(s)

FROM tableName

WHERE [condition]

We want to use the EMPLOYEES table to demonstrate how to use this clause. The table has the following data: [73]

ID	NAME	ADDRESS	AGE	SALARY
2	Mercy	Mercy32	25	3500.00
3	Joel	Joel42	30	4000.00
4	Alice	Alice442	31	2500.00
5	Nicholas	nicoh442	45	5000.00
6	Milly	mil342	32	2000.00
7	Grace	gra361	35	4000.00

The following query will help us fetch the first 2 rows from the table:

SELECT TOP 2 * FROM EMPLOYEES;

Note that the command given above will only work in SQL Server. If you are using MySQL Server, use the LIMIT clause as shown

below:

SELECT * FROM EMPLOYEES

LIMIT 2;[74]

```
mysql> SELECT * FROM EMPLOYEES
    -> LIMIT 2;
+-----+---------+-----------+------+-----------+
| ID  | NAME    | ADDRESS   | AGE  | SALARY    |
+-----+---------+-----------+------+-----------+
|  2  | Mercy   | Mercy32   |  25  | 3500.00   |
|  3  | Joel    | Joel42    |  30  | 4000.00   |
+-----+---------+-----------+------+-----------+
2 rows in set (0.37 sec)

mysql>
```

Only the first two records of the table are returned.

If you are using an Oracle Server, use the ROWNUM with SELECT clause, as shown below:

SELECT * FROM EMPLOYEES

WHERE ROWNUM <= 2;

ORDER BY Clause

This clause helps us sort our data, either in ascending or descending order. The sorting can be done while relying on one or more columns. In most databases, the results are sorted in an ascending order by default.

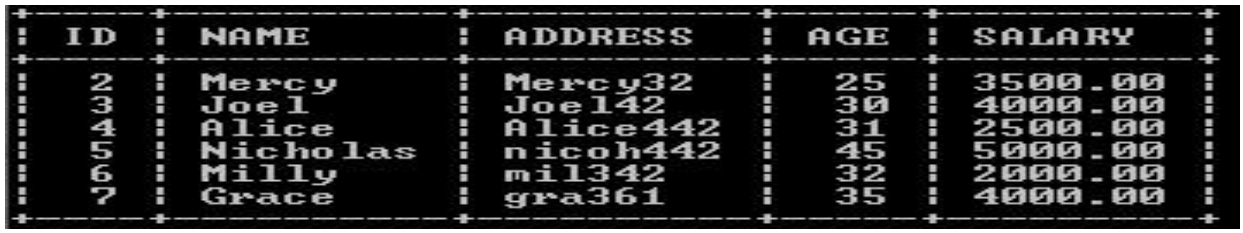
The ORDER BY clause uses the syntax below:

SELECT columns_list

FROM tableName

[WHERE condition]

In the ORDER BY clause, you may use one or more columns. However, you must ensure that the column you choose to sort the data is in the column list. Again, we will use the EMPLOYEES table with the data shown below: [75]



ID	NAME	ADDRESS	AGE	SALARY
2	Mercy	Mercy32	25	3500.00
3	Joel	Joel42	30	4000.00
4	Alice	Alice442	31	2500.00
5	Nicholas	nicoh442	45	5000.00
6	Milly	mil342	32	2000.00
7	Grace	gra361	35	4000.00

Now, we need to use the NAME and SALARY columns to sort the data in ascending order. The following command will help us achieve this:

```
SELECT * FROM EMPLOYEES
ORDER BY NAME, SALARY;
```

The query will return the following result: [76]

```
mysql> SELECT * FROM EMPLOYEES
    ->     ORDER BY NAME, SALARY;
```

ID	NAME	ADDRESS	AGE	SALARY
4	Alice	Alice442	31	2500.00
7	Grace	gra361	35	4000.00
3	Joel	Joel42	30	4000.00
2	Mercy	Mercy32	25	3500.00
6	Milly	mil342	32	2000.00
5	Nicholas	nicoh442	45	5000.00

```
6 rows in set (0.32 sec)

mysql>
```

We can also use the SALARY column to sort the data in descending order:

```
SELECT * FROM EMPLOYEES
```

ORDER BY SALARY DESC;

This is the result:

```
mysql> SELECT * FROM EMPLOYEES
    ->    ORDER BY SALARY DESC;
+----+----------+-----------+-----+----------+
| ID | NAME     | ADDRESS   | AGE | SALARY   |
+----+----------+-----------+-----+----------+
|  5 | Nicholas | nicoh442  |  45 | 5000.00  |
|  3 | Joel     | Joel42    |  30 | 4000.00  |
|  7 | Grace    | gra361    |  35 | 4000.00  |
|  2 | Mercy    | Mercy32   |  25 | 3500.00  |
|  4 | Alice    | Alice442  |  31 | 2500.00  |
|  6 | Milly    | mil342    |  32 | 2000.00  |
+----+----------+-----------+-----+----------+
6 rows in set (0.05 sec)

mysql>
```

https://www.tutorialspoint.com/sql/sql-top-clause.htm

GROUP BY Clause

This clause is used together with the SELECT statement to group data that is related together, creating groups. The GROUP BY clause should follow the WHERE clause in SELECT statements, and it should precede the ORDER BY clause.

The following is the syntax:

SELECT column_1, column_2

FROM tableName

WHERE [conditions]

GROUP BY column_1, column_2

ORDER BY column_1, column_2

Let's use the EMPLOYEES table with the data given below: [77]

ID	NAME	ADDRESS	AGE	SALARY
2	Mercy	Mercy32	25	3500.00
3	Joel	Joe142	30	4000.00
4	Alice	Alice442	31	2500.00
5	Nicholas	nicoh442	45	5000.00
6	Milly	mil342	32	2000.00
7	Grace	gra361	35	4000.00

If you need to get the total SALARY of every customer, just run the following command:

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

The DISTINCT Keyword

This keyword is used together with the SELECT statement to help eliminate duplicates and allow the selection of unique records.

This is because there comes a time when you have multiple duplicate records in a table and your goal is to choose only the unique ones. The DISTINCT keyword can help you achieve this. This keyword can be used with the following syntax:

```
SELECT DISTINCT column_1, column_2,.....column_N
FROM tableName
WHERE [condition]
```

We will use the EMPLOYEES table to demonstrate how to use this keyword. The table has the following data: [78]

ID	NAME	ADDRESS	AGE	SALARY
2	Mercy	Mercy32	25	3500.00
3	Joel	Joel42	30	4000.00
4	Alice	Alice442	31	2500.00
5	Nicholas	nicoh442	45	5000.00
6	Milly	mil342	32	2000.00
7	Grace	gra361	35	4000.00

We have a duplicate entry of 4000 in the SALARY column of the above table. This can be seen after we run the following query:

```
SELECT DISTINCT SALARY FROM EMPLOYEES
ORDER BY SALARY;
```
[79]

```
mysql> SELECT SALARY FROM EMPLOYEES
    -> ORDER BY SALARY;
+----------+
| SALARY   |
+----------+
| 2000.00  |
| 2500.00  |
| 3500.00  |
| 4000.00  |
| 4000.00  |
| 5000.00  |
+----------+
6 rows in set (0.09 sec)

mysql>
```

We can now combine the query with the DISTINCT keyword and see what the query returns: [80]

```
mysql> SELECT DISTINCT SALARY FROM EMPLOYEES
    -> ORDER BY SALARY;
+----------+
| SALARY   |
+----------+
| 2000.00  |
| 2500.00  |
| 3500.00  |
| 4000.00  |
| 5000.00  |
+----------+
5 rows in set (0.07 sec)

mysql>
```

SQL Sub-queries

Until now, we have been executing single SQL queries to perform insert, select, update, and delete functions. However, there is a way to execute SQL queries within the other SQL queries. For instance, you can select the records of all students in the database with an

age greater than a particular student. In this chapter, we shall demonstrate how we can execute sub-queries or queries-within-queries in SQL.

You should have tables labeled “Student” and “Department” with some records.

First, we can retrieve Stacy's age, store it in some variable, and then, using a "where" clause, compare the age in our SELECT query. The second approach is to embed the query that retrieves Stacy's age inside the query that retrieves the ages of all students. The second approach employs a sub-query technique. Have a look at Query 1 to see sub-queries in action.

Query 1

```
Select * From Student
where StudentAge >
(Select StudentAge from Student
where StudName = 'Stacy'
)
```

Notice that in Query 1, we’ve used round brackets to append a sub-query in the “where” clause. The above query will retrieve the records of all students from the “Student” table where the age of the student is greater than the age of “Stacy”. The age of “Stacy” is 20; therefore, in the output, you shall see the records of all students aged greater than 20. The output is the following:

StudID	StudName	StudentAge	StudentGender	DepID
1	Alice	21	Male	2
4	Jacobs	22	Male	5
6	Shane	22	Male	4

7	Linda	24	Female	4
9	Wolfred	21	Male	2
10	Sandy	25	Female	1
14	Mark	23	Male	5
15	Fred	25	Male	2
16	Vic	25	Male	NULL
17	Nick	25	Male	NULL

Similarly, if you want to update the name of all the students with department name “English”, you can do so using the following sub-query:

Query 2

```
Update Student
Set StudName = StudName + ' Eng'
where Student.StudID in (
Select StudID
from Student
Join
Department
On Student.DepID = Department.DepID
where DepName = 'English'
)
```

In the above query, the student IDs of all the students in the English department have been retrieved using a JOIN statement in the sub-query. Then, using an UPDATE statement, the names of all

those students have been updated by appending the string “Eng” at the end of their names. A WHERE statement has been used to match the student IDs retrieved by using a sub-query.

The IFNULL function checks if there is a Null value in a particular Table column. If a NULL value exists, it is replaced by the value passed as the second parameter to the IFNULL function. For instance, the following query will display 50 as the department ID of the students with a null department ID.

Now, if you display the Student’s name along with the name of their Department name, you will see “Eng” appended with the name of the students that belong to the English department.

Exercise 9

Task:

Delete the records of all students from the “Student” table where student’s IDs are less than the ID of “Linda”.

Solution

```
Delete From Student
where StudID <
(Select StudID from Student
where StudName = 'Linda'
)
```

SQL Character Functions

SQL character functions are used to modify the appearance of retrieved data. Character functions do not modify the actual data, but rather perform certain modifications in the way data is represented. SQL character functions operate on string type data. In this chapter, we will look at some of the most commonly used SQL character functions.

Note:

- Concatenation (+)

Concatenation functions are used to concatenate two or more strings. To concatenate two strings in SQL, the '+' operator is used. For example, we can join student names and student genders from the student column and display them in one column, like the following:

```
Select StudName +' '+StudentGender as NameAndGender
from Student
```

- Replace

The replace function is used to replace characters in the output string. For instance, the following query replaces "ac" with "rs" in all student names.

```
Select StudName, REPLACE(StudName, 'ac', 'rs') as ModifiedColumn
From Student
```

The first parameter in the replace function is the column whose value you want to replace; the second parameter is the character sequence which you want to replace, followed by the third parameter which denotes the character sequence you want to insert in place of the old sequence.

- Substring

The substring function returns the number of characters starting from the specified position. The following query displays the first three characters of student names.

```
Select StudName, substring(StudName, 1, 3) as SubstringColumn
From Student
```

- Length

The length function is used to get the length of values of a particular column. For instance, to get the length of names of students in the “Student” table, the following query can be executed:

```
Select StudName, Len(StudName) as NameLength
from Student
```

Note that in the above query, we used the Len() function to get the length of the names; this is because in the SQL server, the Len() function is used to calculate the length of any string.

- IFNULL

The IFNULL function checks if there is a Null value in a particular Table column. If a NULL value exists, it is replaced by the value passed as the second parameter to the IFNULL function. For instance, the following query will display 50 as the department ID of the students with a null department ID.

```
Select Student.DepID, IFNULL(Student.DepID, 50)
from Student
```

- LTRIM

The LTRIM function trims all the empty spaces from the values in the column specified as parameters to the LTRIM function. For instance, if you want to remove all the empty spaces before the names of the students in the “Student” table, you can use the LTRIM query, like the following:

```
Select Student.StudName, LTRIM(Student.StudName)
from Student
```

- RTRIM

The RTRIM function trims all the proceeding empty spaces from the values in the column specified as parameters to the RTRIM function.

For instance, if you want to remove all the empty spaces that come after the names of the students in the “Student” table, you can use the RTRIM query, like the following:

```
Select Student.StudName, RTRIM(Student.StudName)
from Student
```

SQL Constraints

Constraints refer to rules that are applied on the columns of database tables. They help us impose restrictions on the kind of data that can be kept in that table. This way, we can ensure that there is reliability and accuracy of the data in the database.

Constraints can be imposed at column level or at the table level. The column constraints can only be imposed on a single column, while the table level constraints are applied to the entire table.

NOT NULL Constraint

The default setting in SQL is that a column may hold null values. If you don’t want to have a column without a value, you can specify this.

Note that NULL means unknown data rather than no data. This constraint can be defined when you are creating table. Let’s demonstrate this by creating a sample table:

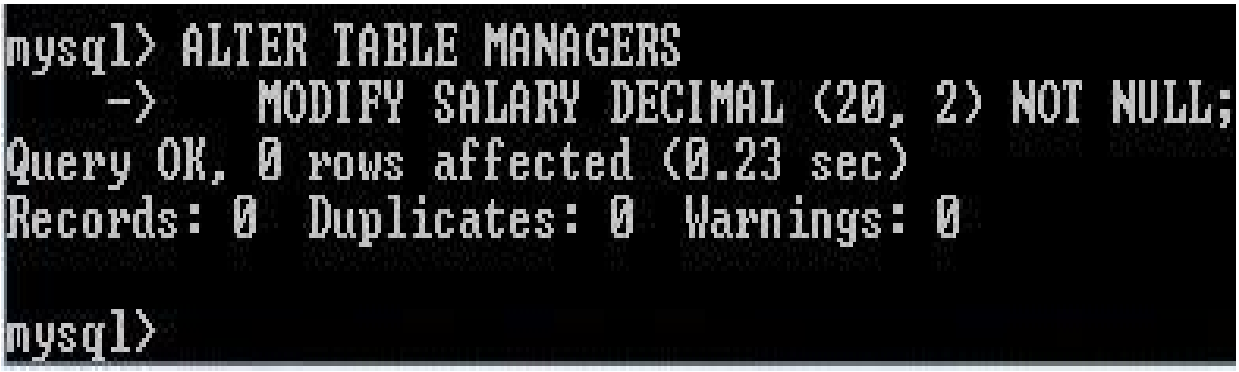
```
CREATE TABLE MANAGERS(
ID INT NOT NULL,
NAME VARCHAR (15) NOT NULL,
DEPT VARCHAR(20) NOT NULL,
SALARY DECIMAL (20, 2),
PRIMARY KEY (ID)
);
```

Above, we have created a table named MANAGERS with 4 columns, and the NOT NULL constraint has been imposed on three of these columns. This means that you must specify a value for each of these columns with the constraint; otherwise, an error will be raised.

Note that we did not impose the NOT NULL constraint to the SALARY column of our table. It is possible for us to impose the constraint on the column even though it has already been created.

ALTER TABLE MANAGERS

MODIFY SALARY DECIMAL (20, 2) NOT NULL; [82]



Now, the column cannot accept a null value.

Default Constraint

This constraint will provide a default value to the column if a value for the column is not specified in the INSERT INTO column.

Consider the example given below:

CREATE TABLE SUPPLIERS(

ID INT NOT NULL,

NAME VARCHAR (15) NOT NULL,

ADDRESS CHAR (20) ,

ARREARS DECIMAL (16, 2) DEFAULT 6000.00,

PRIMARY KEY (ID)

);

Suppose you had already created the table but you needed to add a default constraint on the ARREARS column. You can do it like this:

MODIFY ARREARS DECIMAL (16, 2) DEFAULT 6000.00;

This will update the table, and a default constraint will be created for the column.

If you no longer need this default constraint to be in place, you can drop it by running the command given below:

ALTER TABLE SUPPLIERS

ALTER COLUMN ARREARS DROP DEFAULT;

Unique Constraint

This constraint helps us avoid the possibility of having two or more records with similar values in one column. In the “employees” table, for example, we may need to prevent two or more employees from sharing the same ID.

The following SQL Query shows how we can create a table named “STUDENTS”. We will impose the UNIQUE constraint on the “ADMISSION” column:

CREATE TABLE STUDENTS(

ADMISSION INT NOT NULL UNIQUE,

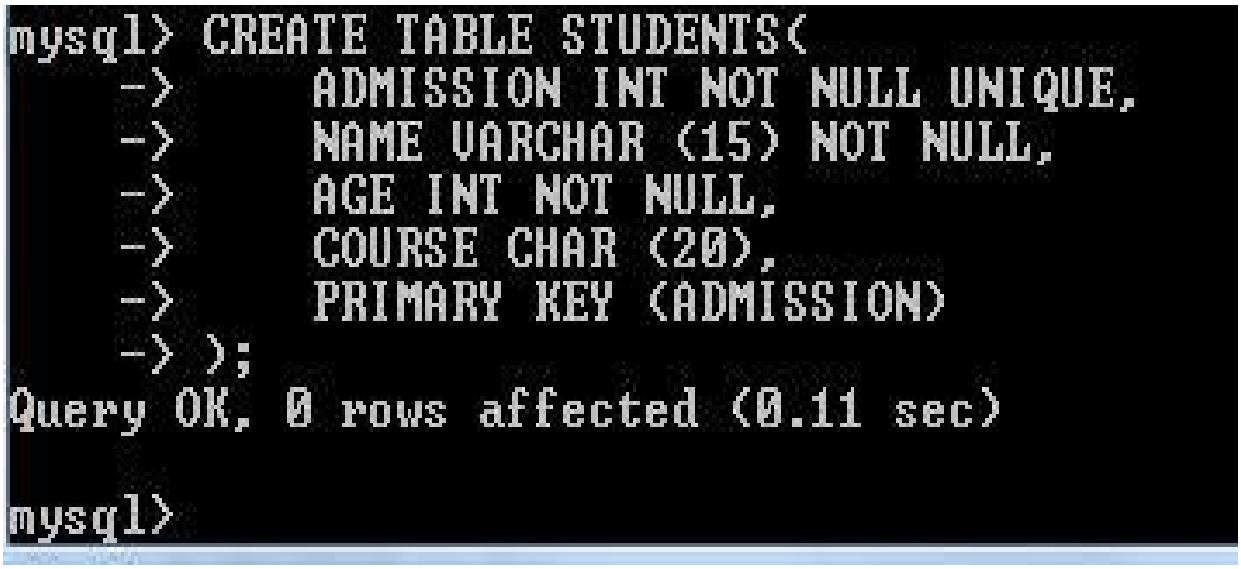
NAME VARCHAR (15) NOT NULL,

AGE INT NOT NULL,

COURSE CHAR (20),

PRIMARY KEY (ADMISSION)

); [83]



Properly created indexes enhance efficiency in large databases. The selection of fields on which to create the index depends on the SQL queries that you use frequently.

Perhaps you had already created the STUDENTS table without the UNIQUE constraint. The constraint can be added on the ADMISSION column by running the following command:

ALTER TABLE STUDENTS

MODIFY ADMISSION INT NOT NULL UNIQUE;

This is demonstrated below:

ALTER TABLE STUDENTS

ADD CONSTRAINT uniqueConstraint UNIQUE(ADMISSION, AGE);

The constraint has been given the name “uniqueConstraint” and assigned two columns, ADMISSION and AGE.

Anytime you need to delete the constraint, run the following command combining the ALTER and DROP commands:

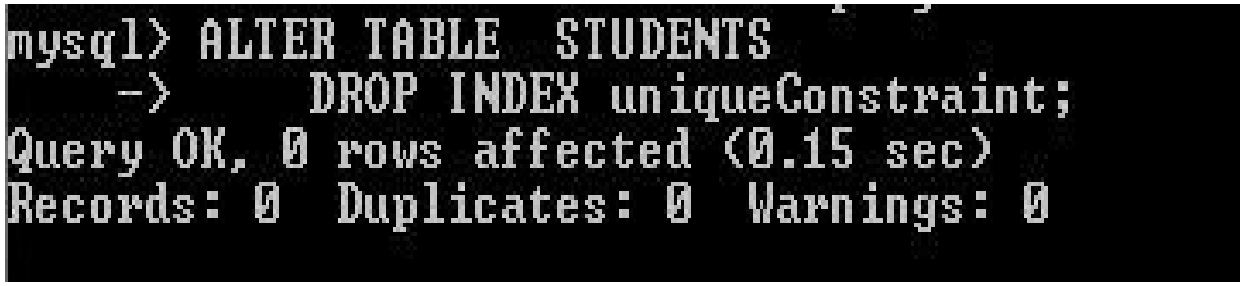
ALTER TABLE STUDENTS

DROP CONSTRAINT uniqueConstraint;

The constraint will be deleted from the two columns. For MySQL users, the above command will not work. Just run the command given below:

ALTER TABLE STUDENTS

DROP INDEX uniqueConstraint; [84]



Primary Key

The primary key constraint helps us identify every row uniquely. The column designated as the primary key must have unique values. Also, the column is not allowed to have NULL values.

Each table is allowed to only have one primary key, and this may be made up of single or multiple fields. When we use multiple fields as the primary key, they are referred to as a composite key. If a column for a table is defined as the primary key, then no two records will share same value in that field.

A primary key is a unique identifier. In the STUDENTS table, we can define the ADMISSION to be the primary key since it identifies each student uniquely. No two students should have the same ADMISSION number. Here is how the attribute can be created:

CREATE TABLE STUDENTS(

ADMISSION INT NOT NULL,

NAME VARCHAR (15) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ADMISSION)

);

The ADMISSION has been set as the Primary Key for the table. If we describe the table, you will find that the field is the primary key, as shown below: [85]

```
mysql> desc Students;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| ADMISSION | int(11)     | NO   | PRI | NULL    |       |
| NAME      | varchar(15) | NO   |     | NULL    |       |
| AGE       | int(11)     | NO   |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
3 rows in set (0.42 sec)

mysql>
```

In the “Key” field above, the “PRI” indicates that the ADMISSION field is the primary key.

You may want to impose the primary key constraint on a table that already exists. This can be done by running the command given below:

ALTER TABLE STUDENTS

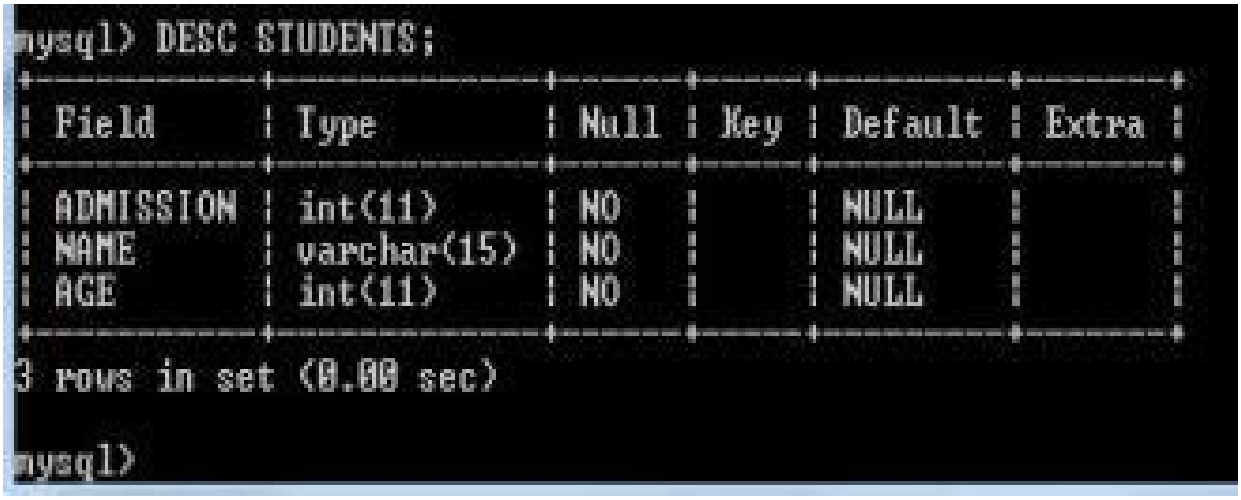
ADD CONSTRAINT PK_STUDADM PRIMARY KEY (ADMISSION, NAME);

In the above command, the primary key constraint has been given the name “PK_STUDADM” and assigned to two columns namely ADMISSION and NAME. This means that no two rows will have the same value for these columns.

The primary key constraint can be deleted from a table by executing the command given below:

ALTER TABLE STUDENTS DROP PRIMARY KEY;

After running the above command, you can describe the STUDENTS table and see whether it has any primary key: [86]



This shows that the primary key was dropped successfully.

Foreign Key

This constraint is used to link two tables together. Sometimes, it is referred to as a referencing key. The foreign key is simply a column or a set of columns that match a Primary Key in another table.

Consider the tables with the structures given below:

STUDENTS table:

CREATE TABLE STUDENTS(

ADMISSION INT NOT NULL,

NAME VARCHAR (15) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ADMISSION)

);

FEE table:

CREATE TABLE FEE (

ID INT NOT NULL,

DATE DATETIME,

STUDENT_ADM INT references STUDENTS (ADMISSION),

AMOUNT float,

PRIMARY KEY (ID)

);[87]

In the FEE table, the STUDENT_ADM field is referencing the ADMISSION field in the STUDENTS table. This makes the STUDENT_ADM column of FEE table a foreign key.

If we had created the FEE table without the foreign key, we could've still added it with the following command:

ALTER TABLE FEES

ADD FOREIGN KEY (STUDENT_ADM) REFERENCES STUDENTS (ADMISSION);

The foreign key will be added to the table. If you need to delete the foreign key, run the following command:

ALTER TABLE FEE

DROP FOREIGN KEY;

The foreign key will then be removed from the table.

CHECK Constraint

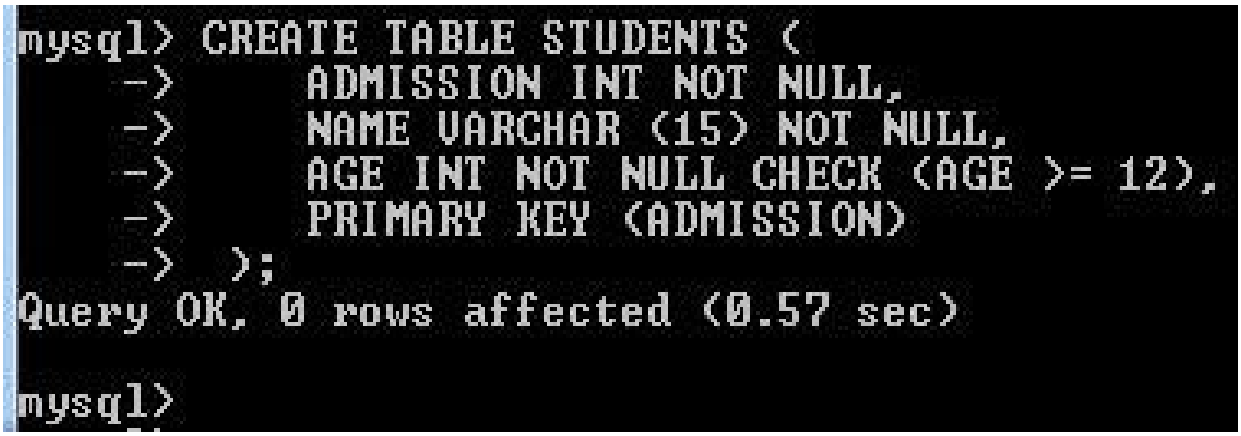
This constraint is used to create a condition that will validate the values that are entered into a record. If the condition becomes false, the record is violating the constraint, so it will not be entered into the table.

We want to create a table called STUDENTS and we don't want to have any student who is under 12 years of age. If the student

doesn't meet this constraint, they will not be added to the table. The table can be created as follows:

```
CREATE TABLE STUDENTS (
ADMISSION INT NOT NULL,
NAME VARCHAR (15) NOT NULL,
AGE INT NOT NULL CHECK (AGE >= 12),
PRIMARY KEY (ADMISSION)
);
```

[88]



The table has been created successfully.

If you had already created the STUDENTS table without the constraint but then needed to implement it, run the command given below:

```
ALTER TABLE STUDENTS
MODIFY AGE INT NOT NULL CHECK (AGE >= 12 );
```

The constraint will be added to the column AGE successfully.

It is also possible for you to assign a name to the constraint. This can be done using the below syntax:

```
ALTER TABLE STUDENTS
ADD CONSTRAINT checkAgeConstraint CHECK(AGE >= 12);
```

https://www.tutorialspoint.com/sql/sql-index.htm

INDEX Constraint

An INDEX helps us quickly retrieve data from a database. To create an index, we can rely on a column or a group of columns in the database table. Once the index has been created, it is given a ROWID for every row before it can sort the data.

Properly created indexes enhance efficiency in large databases. The selection of fields on which to create the index depends on the SQL queries that you use frequently.

Suppose we created the following table with three columns:

```
CREATE TABLE STUDENTS (
ADMISSION INT NOT NULL,
NAME VARCHAR (15) NOT NULL,
AGE INT NOT NULL CHECK (AGE >= 12),
PRIMARY KEY (ADMISSION)
);
```

We can then use the below syntax to implement an INDEX on one or more columns:

```
CREATE INDEX indexName
ON tableName ( column_1, column_2.....);
```

Now, we need to implement an INDEX on the column named AGE to make it easier for us to search using a specific age. The index can be created as follows:

```
CREATE INDEX age_idx
ON STUDENTS (AGE);
```

```
mysql> CREATE INDEX age_idx
    ->       ON STUDENTS (AGE);
Query OK, 1 row affected (0.20 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql>
```

If the index is no longer needed, it can be deleted by running the following command:

ALTER TABLE STUDENTS

DROP INDEX age_idx; [90]

```
mysql> ALTER TABLE STUDENTS
    ->       DROP INDEX age_idx;
Query OK, 1 row affected (0.13 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql>
```

ALTER TABLE Command

SQL provides us with the ALTER TABLE command that can be used for addition, removal and modification of table columns. The command also helps us to add and remove constraints from tables.

Suppose you had the STUDENTS table with the following data: [91]

```
+-------------+-----------+-----+
| ADMISSION   | NAME      | AGE |
+-------------+-----------+-----+
|        3420 | NICHOLAS  |  10 |
|        1234 | john      |  32 |
|        3456 | mercy     |  23 |
+-------------+-----------+-----+
```

ALTER TABLE STUDENTS ADD COURSE VARCHAR(10); [92]

```
mysql> ALTER TABLE STUDENTS ADD COURSE VARCHAR(10);
Query OK, 3 rows affected (0.15 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
```

When we query the contents of the table, we get the following:[93]

```
+-------------+-------------+-------+----------+
| ADMISSION | NAME      | AGE | COURSE |
+-------------+-------------+-------+----------+
|      3420 | NICHOLAS |  10 | NULL   |
|      1234 | john      |  32 | NULL   |
|      3456 | mercy     |  23 | NULL   |
+-------------+-------------+-------+----------+
```

This shows that the column has been added and each record has been assigned a NULL value in that column.

To change the data type for the COURSE column from VarChar to Char, execute the following command:

ALTER TABLE STUDENTS MODIFY COLUMN COURSE Char(1); [94]

```
mysql> ALTER TABLE STUDENTS MODIFY COLUMN COURSE Char(1);
Query OK, 3 rows affected (0.13 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
```

We can combine the ALTER TABLE command with the DROP TABLE command to delete a column from a table. To delete the COURSE column from the STUDENTS table, we run the following command:

ALTER TABLE STUDENTS DROP COLUMN COURSE; [95]

```
mysql> ALTER TABLE STUDENTS DROP COLUMN COURSE;
Query OK, 3 rows affected (0.18 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
```

We can then view the table details via the describe command to see whether the column was dropped successfully: [96]

```
mysql> desc students;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| ADMISSION | int(11)     | NO   | PRI | NULL    |       |
| NAME      | varchar(15) | NO   |     | NULL    |       |
| AGE       | int(11)     | NO   |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
3 rows in set (0.03 sec)

mysql>
```

The above figure shows that the column was dropped successfully.

CONCLUSION AND NEXT STEPS

I hope that you enjoyed the book and were able to get as much information out of it as possible. If you like, feel free to go back to any other sections of the book and bounce around to sharpen your skills.

Product Review and Feedback

Rather than just giving you a standard conclusion, I'd like to give you some suggestions on the next steps that you can take to continue learning SQL. But first, if you have some feedback, hop over to the Amazon product page for this book and leave an honest review, or email me directly at jaschulz0705 gmail.com.

More Database Samples

If you're interested in working with more databases, then I have a few links that you will be interested in. You can download the Chinook and Northwind databases from the following links and begin working with those.

Go ahead and click this link to download the Chinook database. Once you're there, click the arrow on the top-right to download a zip file. Once you've done that, extract the zip file to a location that is easy to access on your computer. Now, open the folder and drag and drop either the Chinook_SqlServer.sql file or Chinook_SqlServer_AutoIncrementPKs.sql into SSMS and click the 'Execute' button. You'll then have a full Chinook database.

In addition to the Chinook database, you can click this link to download the Northwind database. Once here, click the arrow on the top-right to download another zip file. Then, extract the zip file from here and place it with the rest of your SQL backup files. Then, restore the database since it's a Northwind.bak file (database backup file). You can also refer to the previous sections for

database restores and the fundamentals of SSMS to assist with these tasks!

Keep Learning

Dig into the AdventureWorks, Company_Db, Chinook and Northwind databases by running queries and understanding the data. Part of learning SQL is understanding the data within a database, too.

To protect databases, SQL has a security scheme that lets you specify which database users can view specific information from. This scheme also allows you to set what actions each user can perform. This security scheme (or model) relies on authorization identifiers. As you've learned in the second chapter, authorization identifiers are objects that represent one or more users that can access/modify the information inside the database.

More References

MTA 98-364 Certification

You can find the information for the certification here: https://www.microsoft.com/en-us/learning/exam-98-364.aspx. Once you're there, be sure to expand on the topics in the 'Skills Measured' section, as your database knowledge in these areas will be put to the test.

That's all for now.

REFERENCES

1keydata.com. (2019). *SQL - CREATE VIEW Statement | 1Keydata*. [online] Available at: https://www.1keydata.com/sql/sql-create-view.html [Accessed 3 Feb. 2019].

Chartio. (2019). *How to Alter a Column from Null to Not Null in SQL Server*. [online] Available at: https://chartio.com/resources/tutorials/how-to-alter-a-column-from-null-to-not-null-in-sql-server/ [Accessed 3 Feb. 2019].

Docs.microsoft.com. (2019). *Primary and Foreign Key Constraints - SQL Server*. [online] Available at: https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-2017 [Accessed 3 Feb. 2019].

Sites.google.com. (2019). *DDL Commands - Create - Drop - Alter - Rename - Truncate - Programming Languages*. [online] Available at: https://sites.google.com/site/prgimr/sql/ddl-commands---create---drop---alter [Accessed 3 Feb. 2019].

Techonthenet.com. (2019). *SQL: UNION ALL Operator*. [online] Available at: https://www.techonthenet.com/sql/union_all.php [Accessed 3 Feb. 2019].

1keydata.com. (2019). *SQL - RENAME COLUMN | 1Keydata*. [online] Available at: https://www.1keydata.com/sql/alter-table-rename-column.html [Accessed 3 Feb. 2019].

Docs.microsoft.com. (2019). *Create Check Constraints - SQL Server*. [online] Available at: https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-check-constraints?view=sql-server-2017 [Accessed 3 Feb. 2019].

Docs.microsoft.com. (2019). *Create Primary Keys - SQL Server*. [online] Available at: https://docs.microsoft.com/en-us/sql/relational-

databases/tables/create-primary-keys?view=sql-server-2017 [Accessed 3 Feb. 2019].

Docs.microsoft.com. (2019). *Delete Tables (Database Engine) - SQL Server*. [online] Available at: https://docs.microsoft.com/en-us/sql/relational-databases/tables/delete-tables-database-engine?view=sql-server-2017 [Accessed 3 Feb. 2019].

Docs.microsoft.com. (2019). *Modify Columns (Database Engine) - SQL Server*. [online] Available at: https://docs.microsoft.com/en-us/sql/relational-databases/tables/modify-columns-database-engine?view=sql-server-2017 [Accessed 3 Feb. 2019].

query?, H., M, D., K., R., Singraul, D., Singh, A. and kor, p. (2019). *How to change a table name using an SQL query?*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/886786/how-to-change-a-table-name-using-an-sql-query [Accessed 3 Feb. 2019].

SQL Joins Explained. (2019). *SQL Join Types — SQL Joins Explained*. [online] Available at: http://www.sql-join.com/sql-join-types/ [Accessed 3 Feb. 2019].

Techonthenet.com. (2019). *SQL: ALTER TABLE Statement*. [online] Available at: https://www.techonthenet.com/sql/tables/alter_table.php [Accessed 3 Feb. 2019].

Techonthenet.com. (2019). *SQL: GROUP BY Clause*. [online] Available at: https://www.techonthenet.com/sql/group_by.php [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL DEFAULT Constraint*. [online] Available at: https://www.tutorialspoint.com/sql/sql-default.htm [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL INDEX Constraint*. [online] Available at: https://www.tutorialspoint.com/sql/sql-index.htm

[Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL INNER JOINS*. [online] Available at: https://www.tutorialspoint.com/sql/sql-inner-joins.htm [Accessed 3 Feb. 2019].

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

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

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

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

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

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

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

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

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

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

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

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

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

Conclusion

The next step is to get out there and start making your own sketches! Go to your local hobby store to get some ideas or go to the community to see what new projects you might want to try. After you have an idea where you might want to go next, (robots are pretty fun!) join the community! Seriously, it is a lot of fun to build projects with friends and compare them with each other. If you feel like you don't know where to start, don't worry! There are many online sources that share coding and techniques to improve your game. Many online sites also have forums specifically tailored to helping people like you learn and show off what they have done. It is also a fantastic way to learn and grow as a hobbyist.

If you want to get started but are feeling strapped for cash, there are options. Like we've said above, there are fairly cheap modules for purchase on the Arduino site and others. Also, cheaper programming languages are available, and some are even free. Learning these languages are actually easier than you would think. If you know one programming language, the rest are easier to understand and write. There are also books at your local library that will help you learn how to code. Many libraries offer interlibrary loans, which means that you can learn about programming from books from all around you! Best of all, learning from a book from the library is absolutely free! If you have any questions about this, remember that you can go online and find others who have worked with Arduino and know how to get you started.

Arduino board can be programmed to light or fade a LED etc. The syntax used in Arduino programming is similar to C++. If you are

good at C++, then programming in Arduino will be easy for you. The variables in Arduino are initialized within the setup() function. The loop() section has the block to be run repeatedly. When working with Arduino pins, you must specify the pin you need to work with. The pins are normally identified with numbers as each has a unique number. After getting the board, you have to setup Arduino IDE on your computer. This is where you will be writing your Arduino code before uploading bit to the board. Arduino code is commonly known as a sketch. You must get a source of power. However, some board types must be configured to allow power to be drawn from a computer. The effect of a sketch on the board will be seen after uploading it to the board, in which one has to click the Upload button.

There are many sensors and additions to each Arduino, so make sure you check out which you would like to employ. The great part of owning an Arduino is that you'll get the chance to try many experiments. You're not just limited to what the sensors can read, either. Come up with some ways on your own to make the machines work for you. Try programming simple requests first—like setting up blinking lights or figuring out how Arduino can monitor inputs—and see what you can do from there. I have included many Arduino codes for you to use, but feel free to find some of your own! There are many guides for help.

You can also check out some more advanced concepts we didn't have a chance to touch on here such as headers, classes, changing the clock speed for the chip, adding cores, adding libraries, there is so much that you can do with this chip, it really is incredible. Pick a direction that interests you and see where it takes you. I hope that this guide has offered you some small inspiration to go

out there and try new things and see what your sketch designing skills are capable of.

Finally, thank you for finishing this book! Arduinos are a fun way to get started on your programming journey. Since you've purchased this book, we hope you've grown, and if you found this book useful in any way, a review on Amazon is always appreciated!

References

Arduino Reference. (2019). Retrieved from https://www.arduino.cc/reference/en/language/structure/comparison-operators/lessthan/

What is an Arduino? - learn.sparkfun.com. (2019). Retrieved from https://learn.sparkfun.com/tutorials/what-is-an-arduino/all#the-arduino-family