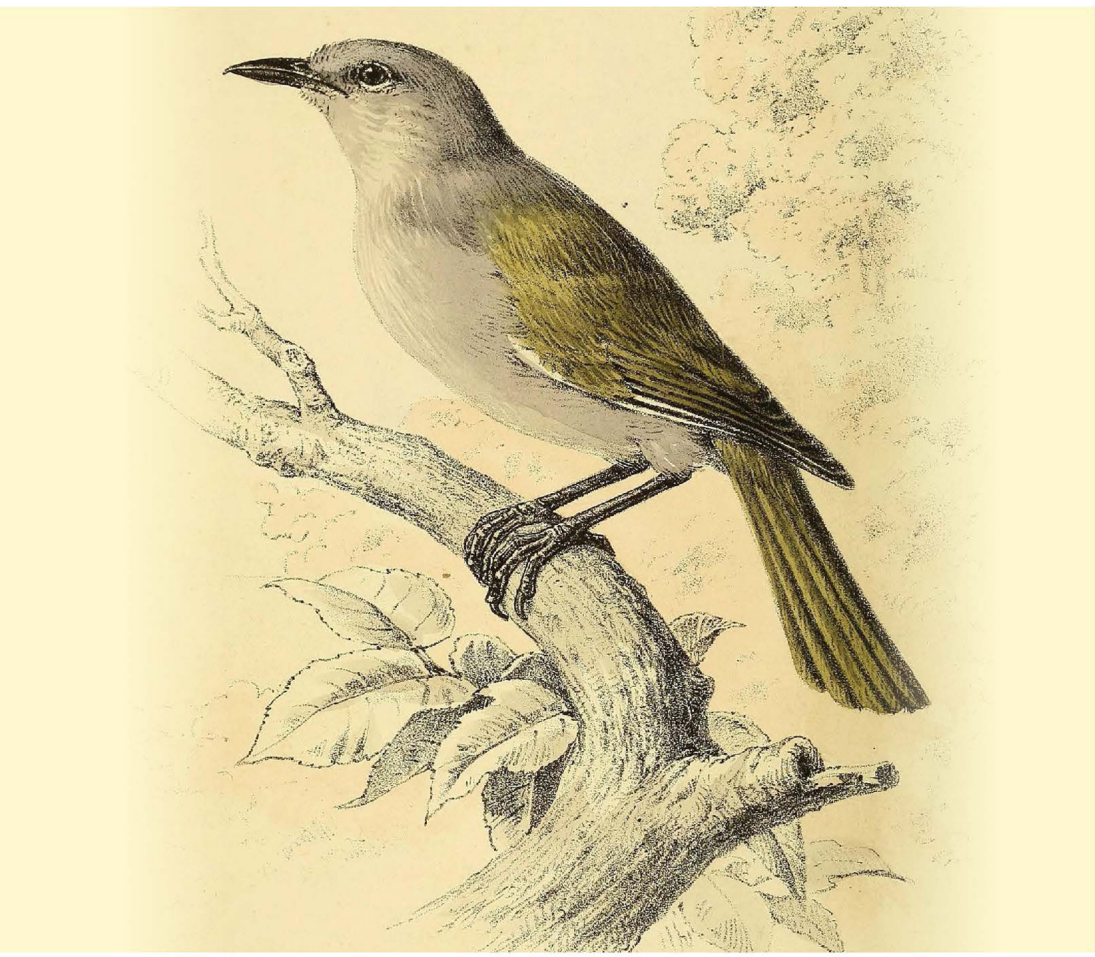
MASTERING COMPUTER SCIENCE

Mastering SQL

A Beginner's Guide



EDITED BY

Sufyan bin Uzayr

CRC Press
Taylor & Francis Group

Mastering SQL

Mastering SQL helps the readers gain a firm understanding of the Structured Query Language.

Structured Query Language, more often known as SQL, is the de facto standard language for working with databases. It is a specialised language for handling data-related tasks like creating a database, putting information into tables, modifying and extracting that information, and much more. MySQL, PostgreSQL, Oracle, SQL light, etc. are only a few examples of SQL implementations.

SQL is a fast and efficient database system. SQL allows for the rapid and efficient retrieval of huge numbers of data entries from a database. It's a relational database. Thus, the data is described in a more orderly fashion than in an unstructured database like MongoDB. Insertions, deletions, inquiries, manipulations, and computations of data through analytical queries in a relational database may all be performed in a matter of seconds.

With *Mastering SQL*, learning SQL becomes straightforward; using this book and resource will undoubtedly help readers advance their careers.

About the Series

Mastering Computer Science covers a wide range of topics, spanning from programming languages to modern-day technologies and frameworks. The series has a special focus on beginner-level content and is presented in an easy-to-understand manner, comprising:

- Crystal-clear text, spanning various topics sorted by relevance,
- Special focus on practical exercises, with numerous code samples and programs,
- A guided approach to programming, with step-by-step tutorials for the absolute beginners,
- Keen emphasis on real-world utility of skills, thereby cutting the redundant and seldom-used concepts and focusing instead on industry-prevalent coding paradigm,
- A wide range of references and resources, to help both beginner- and intermediate-level developers gain the most out of the books.

The *Mastering Computer Science* series of books start from the core concepts and then quickly move on to industry-standard coding practices, to help learners gain efficient and crucial skills in as little time as possible. The books assume no prior knowledge of coding, so even the absolute newbie coders can benefit from this series.

The *Mastering Computer Science* series is edited by Sufyan bin Uzayr, a writer and educator with over a decade of experience in the computing field.

For more information about this series, please visit https://www.routledge.com/Mastering-Computer-Science/book-series/MCS

Mastering SQL
A Beginner's Guide

Edited by
Sufyan bin Uzayr

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

First Edition published 2024
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

Library of Congress Cataloging-in-Publication Data
Names: Bin Uzayr, Sufyan, editor.
Title: Mastering SQL : a beginner's guide / edited by Sufyan bin Uzayr.
Description: Boca Raton : CRC Press, 2024. | Series: Mastering computer science |
Includes bibliographical references and index.
Identifiers: LCCN 2023007431 (print) | LCCN 2023007432 (ebook) |
ISBN 9781032415062 (paperback) | ISBN 9781032415093 (hardback) |
ISBN 9781003358435 (ebook)
Subjects: LCSH: SQL (Computer program language) |
Database management. Classification: LCC QA76.73.S67 M324 2024 (print) |
LCC QA76.73.S67 (ebook) | DDC 005.75/6—dc23/eng/20230502
LC record available at https://lccn.loc.gov/2023007431
LC ebook record available at https://lccn.loc.gov/2023007432

ISBN: 9781032415093 (hbk)
ISBN: 9781032415062 (pbk)
ISBN: 9781003358435 (ebk)

DOI: 10.1201/9781003358435

Typeset in Minion Pro
by codeMantra

For Mom

Contents

About the Editor

Sufyan bin Uzayr is a writer, coder, and entrepreneur with over a decade of experience in the industry. He has authored several books in the past, pertaining to a diverse range of topics, ranging from History to Computers/IT.

Sufyan is the Director of Parakozm, a multinational IT company specialising in EdTech solutions. He also runs Zeba Academy, an online learning and teaching vertical with a focus on STEM fields.

Sufyan specialises in a wide variety of technologies, such as JavaScript, Dart, WordPress, Drupal, Linux, and Python. He holds multiple degrees, including ones in management, IT, literature, and political science.

Sufyan is a digital nomad, dividing his time between four countries. He has lived and taught in universities and educational institutions around the globe. He takes a keen interest in technology, politics, literature, history, and sports, and in his spare time, he enjoys teaching coding and English to young students.

Learn more at sufyanism.com

Acknowledgements

There are many people who deserve to be on this page, and this book would not have come into existence without their support. That said, some names deserve a special mention, and I am genuinely grateful to

- My parents, for everything they have done for me.
- The Parakozm team, especially Divya Sachdeva, Jaskiran Kaur, and Simran Rao, for offering great amounts of help and assistance during the book-writing process.
- The CRC team, especially Sean Connelly and Danielle Zarfati, for ensuring that the book's content, layout, formatting, and everything else remain perfect throughout.
- Reviewers of this book for going through the manuscript and providing their insight and feedback.
- Typesetters, cover designers, printers, and everyone else for their part in the development of this book.
- All the folks associated with Zeba Academy, either directly or indirectly, for their help and support.
- The programming community in general, and the web development community in particular, for all their hard work and efforts.

– Sufyan bin Uzayr

Zeba Academy – Mastering Computer Science

The 'Mastering Computer Science' series of books are authored by the Zeba Academy team members, led by Sufyan bin Uzayr, consisting of

- Divya Sachdeva.
- Jaskiran Kaur.
- Simran Rao.
- Aruqqa Khateib.
- Suleymen Fez.
- Ibbi Yasmin.
- Alexander Izbassar.

Zeba Academy is an EdTech venture that develops courses and content for learners primarily in STEM fields, and offers educational consulting and mentorship to learners and educators worldwide.

Additionally, Zeba Academy is actively engaged in running IT schools in the CIS countries and is currently working in partnership with numerous universities and institutions.

For more info, please visit https://zeba.academy

CHAPTER 1

Basics about SQL

IN THIS CHAPTER

- Basics about SQL
- Purpose of SQL
- Features of SQL
- SQL and its components
- Introduction to NoSQL
- Advantages and disadvantages

Data is nowadays the groundwork of any professional. Enterprise computing is the pinnacle of data-driven organisations. The significance of organised data storage is undeniably highlighted. Every day, we generate vast amounts of data. At our current rate, more than 2.5 quintillion bytes of data are created each day. It is anticipated that by 2025, the world would generate ~463 exabytes of data every day. This demonstrates the importance of data management, which is where databases come into play. We all know that data originates from various sources and is unstructured. A database is a place where this data can be stored in an organised manner. A database consists of rows and columns. Data is classified and kept in the form of tables here. A database is a structured collection of data that can be retrieved quickly. Database Management Systems (DBMS) are used to manage these databases (Figure 1.1).

DOI: 10.1201/9781003358435-1

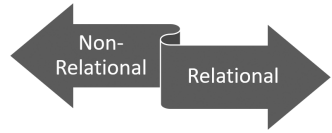


FIGURE 1.1 Types of databases.

It is therefore important to have a basic understanding of this hierarchical data storage and retrieval paradigm. On the other hand, a beginner will surely feel lost in the vast amount of knowledge available in the wild ecosphere. Structured Query Language (SQL) is a query language for working with relational databases. We can use SQL to update, delete, and retrieve data from a database. A table is used to define the data/object. These tables consist of rows and columns and are uniquely identified by their field names.

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)

It takes into account the relationship between tables using primary keys, foreign keys, and indexes. This gives it a significant advantage over DBMS in terms of data retrieval and storage. It is used in enterprises to store huge amounts of complex data.

KEYS IN SQL

Keys are a fundamental part of the relational database paradigm. They are used to define and identify relationships between tables, as well as to identify any record or row of data in a table.

- A **super key** is used to identify rows in a table.
- A **primary key** is a collection of one or more table fields that uniquely identify a record in a database table. It is unable to accept null or duplicate values. A table contains only one primary key.
- **Alternate Key**: A field or set of fields that uniquely identify each row in the database. A table can have various primary keys, but only one can be set as the primary key. Another key is any key that is not a primary key.
- A **unique key** is a collection of one or more fields that uniquely identify a record in a database. It is comparable to a primary key except that it can have one null value.

- A **foreign key** is a field in one table that is a primary key in another table. It creates a relationship between two tables while ensuring data integrity and navigation.

SQL has been around for more than four decades. Nonetheless, it really has changed massively. Throughout our academics, we all encountered SQL at a certain point. We are all here to make things better by moving away from outdated tools and methods. When we learn a system utilising old methods and procedures, we must internalise it and reconsider its use in modern industry after you become a member of it. SQL is an abbreviation for Structured Query Language. It is a computer language that is commonly used for Relational Database Management Systems (RDBMS) and data manipulation.

Understand that SQL is not going away anytime soon. It has only become stronger as a consequence of corporate objectives such as Microsoft, which has recently made SQL available for Linux as well. The numerous cloud products, such as Microsoft Azure and Amazon Web Services – the two biggest participants in the market – take a much focused approach to the relational database model, and SQL in particular. And, this is only the start of a new epoch. When you enter this period, it's critical that you understand how things function so that you can take the company you own or support to new heights.

WHAT IS STRUCTURED QUERY LANGUAGE (SQL)?

It is a computer language used to perform database operations such as upgrading, adding, eliminating, and building and editing tables in the database and views. SQL is a query language, not a database system. Assume you want to use the SQL language to run queries on the database's data. Any database management system, such as Oracle, MySQL, MongoDB, PostgreSQL, SQL Server, and DB2, must be installed in your systems. The major purpose of this database language is to maintain data in relational database management systems. Data experts utilise this specialised tool to work with structured data (data which is stored in the form of tables). It's also made for RDSMS's stream processing. Among other things, you may rapidly create and manage the database as well as access and alter table rows and columns. In 1986, the ANSI approved this query language as a standard, and in 1987, the ISO followed suit. If you want to work in the field of data science, it is crucial to grasp this query language.

SQL is used by big companies like Facebook, Instagram, and LinkedIn to store data in the backend.

ORIGINS OF STRUCTURED QUERY LANGUAGE (SQL)

In 1970, the eminent computer scientist E.F. Codd wrote an information retrieval titled 'A Relational Model of Data for Large Shared Data Banks'.[1] After learning from E.F. Codd's article, two of the IBM researchers Raymond Boyce and Donald Chamberlin developed the SEQUEL (Structured English Query Language). In 1970, they worked together at IBM's San Jose Research Laboratory to create SQL. Relational Software Inc. created the first SQL around the end of the 1970s, following the ideas of E.F. Codd, Raymond Boyce, and Donald Chamberlin. This SQL was entirely RDBMS-based. In June 1979, Relational Software Inc., which is now known as Oracle Corporation, released the Oracle V2, the first implementation of SQL.

PURPOSE OF STRUCTURED QUERY LANGUAGE (SQL)

SQL is commonly used in data science and analytics lately. The foregoing is among some of the grounds for its widespread need:

- For data professionals and SQL users, the most fundamental uses of SQL are for inserting, updating, and removing data from a relational database.
- SQL can be used to get data from relational database management systems for users and data professionals.
- Additionally, it helps with structured data description.
- Databases and tables can be created, deleted, and managed by SQL users.
- It also facilitates the creation of relational database views, stored procedures, and functions.
- It enables users to define and update the information maintained in a relational database.
- SQL clients can also define access and conditions for table columns, views, and stored procedures.

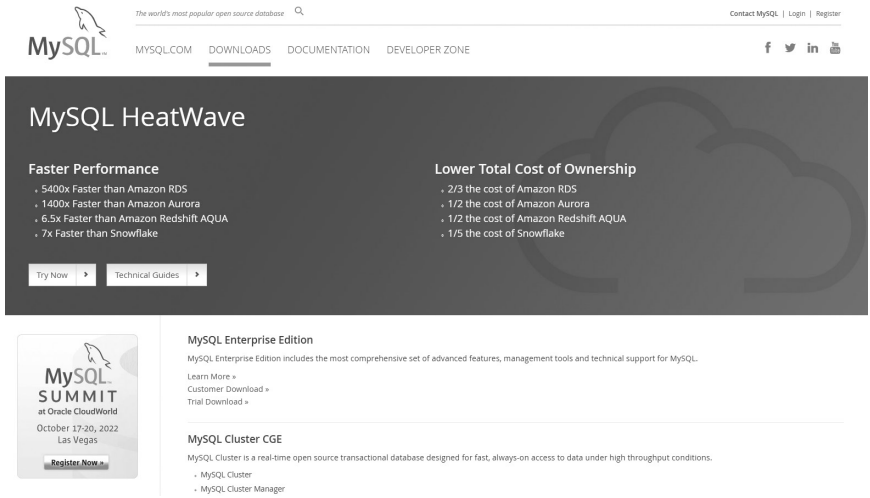


FIGURE 1.2 MySQL installation page.

INSTALLATIONS OF SQL

MySQL is an open-source relational database management system that may be downloaded from the official website at https://dev.mysql.com/downloads. Start MySQL service after it has been installed (Figure 1.2).

OPERATIONS IN STRUCTURED QUERY LANGUAGE (SQL)

When we run a SQL command on a relational database management system, the system will automatically locate the appropriate routine to carry out our request, and the SQL engine will decide how to interpret that command.

The four components of the Structured Query Language procedure are as follows:

- Optimisation Engines.
- SQL Query Engine.
- Classic Query Engine.
- Query Dispatcher.

Data professionals and users can use a classic query engine to maintain non-SQL queries. Figure 1.3 illustrates the SQL architectural style.

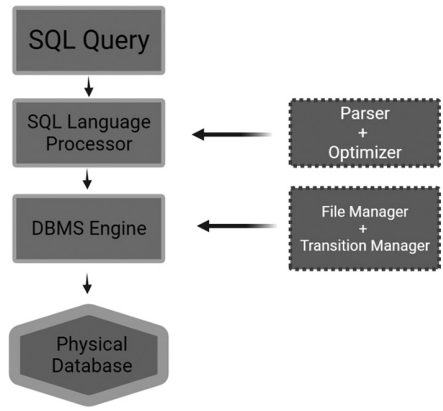


FIGURE 1.3 Framework of SQL.

WHAT ARE THE DIFFERENT TYPES OF FUNCTIONS?

Methods for performing data operations are called functions. SQL provides a number of built-in functions for text concatenation, mathematical calculations, and other tasks.

The following are the different types of SQL functions:

- Aggregated Functions.
- Scalar Functions.

Let's examine each of them separately.

Aggregated Functions

In SQL, an aggregate function adds up a set of values and returns a single result. A handful of the most regularly used Aggregate Functions are listed below:

Function	Description
SUM()	Is used to calculate the total of a set of values.
COUNT()	Gets the total number of rows, either with or without a condition.
AVG()	To calculate the avg value of a numeric column, use this function.
MIN()	The minimal value of a column is returned.
MAX()	Returns a maximum value of a column.
FIRST()	The first value of the column is returned.
LAST()	The final value of the column is returned.

Scalar Functions

Scalar functions in SQL return a single value based on the input value. Some of the most widely used Aggregate Functions are listed below:

Function	Description
LCASE()	Convert string column values to lowercase.
UCASE()	Convert a string column values to Uppercase.
LEN()	Returns the length of the text values in the column.
MID()	Extracts substrings in SQL from column values having String data type.
ROUND()	Rounds off a numeric value to the nearest integer.
NOW()	Return the current system date and time.
FORMAT()	Format how a field must be displayed.

CHARACTERISTICS OF STRUCTURED QUERY LANGUAGE (SQL)

In today's technological environment, regardless of how it appears on the surface, every programme or development tool ends up translating queries and other commands into SQL. The characteristics of SQL are numerous and may be described in greater depth, but we've focused on the most popular and crucial elements that have helped SQL maintain its prominence to this day. Because of these significant capabilities, no other database system has been able to attain the same level of success as SQL.

Foundational Relationships

SQL is a relational database language. The tabular layout of a relational database provides an intuitive user interface, making SQL simple to learn and use. Furthermore, relational models have a strong theoretical foundation that has influenced relational database development and deployment. SQL has become the database language for relational databases as a result of the popularity of the relational paradigm.

High Performance

A significant volume of data is swiftly and efficiently retrieved. Furthermore, simple data manipulation tasks such as inserting, deleting, and modifying data can be completed in a short amount of time. SQL is so fast because a database product must deliver sets of data quickly when requested in order to be successful. Many of the company's brightest individuals labour

around the clock on the query engine to ensure that it generates 'optimal' query strategies that operate quickly.

Scalability

The SQL database is vertically scalable, which means that by adding more RAM, SSDs, or CPUs, you may increase the load on a single server. Because of the way data is stored (connected tables vs unrelated collections), SQL databases can only scale vertically – horizontal scalability is only possible with NoSQL databases.

Authentication and Security

Several security-enhancing features are included in SQL Server, including encrypted communication over SSL/TLS, the Windows Data Protection API (DPAPI) to encrypt data at rest, authentication, and permission. The process of identifying a user or a person based on their login and password is known as authentication. The credentials of SQL Server's users are used to authenticate them. SQL Server has two authentication modes: Windows authentication and mixed-mode authentication.

- Windows authentication is the default authentication technique also known as integrated security due to its tight integration with Windows. To connect to SQL Server, specific Windows user and group accounts are trusted. Additional credentials are not required for users who have already been authenticated.
- Mixed-mode authentication works with both Windows and SQL Server. Mode and mixed-mode keep track of user names and passwords.

Independent Vendors

No new DBMS product has been very successful over the last decade, despite SQL support being available from all major DBMS manufacturers. With minimal conversion work, SQL-based databases and programmes can be moved from one DBMS to another vendor's DBMS. As a result, vendor independence is one of SQL's most fundamental characteristics and a key reason for its early appeal.

Adaptability to a Variety of Computer Systems

Mainframes, PCs, workstations, specialised servers, and even handhelds are all supported by SQL-based database products.

As a result of SQL's feature,

- As applications grow, they can be transferred from single-user or departmental servers to bigger server systems.
- Data from corporate SQL databases can also be extracted and downloaded into departmental or personal databases.
- Before moving to a more expensive multiuser system, a prototype of a SQL-based database application can be constructed on a less expensive personal computer.

Endorsement and Commitment from IBM (DB2)

SQL was established by IBM researchers and has now evolved into a key product based on IBM's flagship DB2 database. SQL is supported on all major IBM platforms, from personal computers through midrange systems to mainframes. As a result of IBM's early efforts, other database and system suppliers followed IBM's lead in the development of SQL and relational databases, and SQL was accepted more swiftly by the market as a result of IBM's widespread support and dedication. The SQL-based products that IBM has produced run on hardware from developing competitors such as HP and Sun, in addition to IBM's own products.

Structure Similar to That of English

SQL is basic and easy to understand because it uses English-like words like create, select, delete, and update. Columns and tables in SQL databases can have long, descriptive names. As a result, the majority of SQL statements have a clear meaning and can be read as natural sentences.

Database Access through Programming

When writing apps, programmers utilise SQL to access databases. Because interactive and programmatic access to the database is made possible by using the same SQL statements, unlike traditional databases that have separate tools for programmatic and unscheduled requests, database

access components of a programme can be tested first with interactive SQL before being embedded into a programme.

Transaction Control Language

The propagation of a change in the database is referred to as a transaction in transaction control language. Transactions are an important part of a database management system, and TCL (Transaction Control Language) is used to manage them. TCL includes commands like commit, rollback, and save point.

Various Viewpoints on Data

A database's creator can use SQL to provide various users with distinct views of the database's structure and content. For example, an organisation's database can be set up so that each user can only access data from his or her own department. Furthermore, data from many database tables can be merged and displayed to the user as a simple row/column table.

Dynamic

One of the most significant advantages of SQL over other static databases is the ability to update and expand a database's structure dynamically, even while users are accessing database material. As a result, SQL offers the most flexibility, allowing online applications to run uninterrupted as a database adapts to changing needs.

Architecture of Client/Server

A client–server relationship is one in which a client (or a group of clients) is connected to a server (one). SQL implementation is a natural fit for applications with distributed client/server systems. A SQL database connects 'front-end' computer systems that focus on user interface with 'back-end' computer systems that focus on database management, allowing each system to perform what it does best. SQL also enables personal computers to act as a front end to network servers or mainframe databases, allowing access to corporate data via a desktop application.

SQL with Java Integration

Integration of SQL with Java has become a prominent focus of SQL development in recent years. To connect the Java language to existing relational databases, Sun Microsystems (the creator of JAVA) released Java Database

Connectivity JDBC (a standard API that allows Java programs to use SQL for database access). It ensured that SQL would continue to be relevant in the emerging Java-based programming environment.

SOME SQL STATEMENTS

The SQL commands are used to create and manage databases. The list contains some of the most used SQL commands:

- **Create:** The function facilitates the creation of the new database, table, table view, and other database objects.
- **Update:** This operation is used to update or change the database's stored data.
- **Delete:** The saved records can be deleted or removed from database tables using this function. It purges one or more tuples at a time from database tables.
- **Select:** This function allows you to access a single or more rows from one or more database tables. This command can also be used with the WHERE clause.
- **Drop:** This function can be used to delete an entire table, table view, or any database item.
- **Insert:** This function facilitates the entry of data or records into database tables. The records can be readily inserted into single or many rows of the table.

ADVANTAGES OF STRUCTURED QUERY LANGUAGE (SQL)

In the world of data science, SQL has grown in popularity thanks to a variety of benefits. It's an excellent query language for users and data professionals to communicate with databases. The advantages or benefits of Structured Query Language are as follows:

- **There Is No Requirement for Programming:** For database management, SQL does not necessitate a huge amount of coding lines. Using simple SQL syntactical rules, we can quickly access and maintain the database. The SQL is user-friendly thanks to these simple guidelines.

- **Query Processing at a High Rate:** SQL queries are used to quickly and effectively access a large amount of data from the database. Less time is required for data operations like insertion, deletion, and updating.
- **Language That Is Standardised:** SQL complies with ISO and ANSI standards, giving its users a uniform platform around the globe.
- **Convenience**: The Structured Query Language can be used on desktop computers, laptops, tablets, and even smartphones.
- **Language That Is Interactive:** The SQL language is simple to learn and comprehend. Because it is a basic query language, we can also use it to communicate with the database. This language is also utilised for receiving instant replies to difficult queries.
- **There Are Multiple Data Views:** The construction of numerous database structure views for diverse database users is also made easier by the SQL language.

DISADVANTAGES OF STRUCTURED QUERY LANGUAGE (SQL)

Despite all of SQL's advantages, it also has certain disadvantages, including the following:

- **Price:** Some SQL versions have a significant operational cost. As a result, some programmers are unable to use Structured Query Language.

  It Has a Difficult User Interface: The Structured Query Language interface's complexity makes it challenging for SQL users to use and administer, which is another important drawback.
- **Database Control in Part:** The corporate regulations are kept under wraps. As a result, data professionals and users who utilise this query language are unable to have complete database control.

DATA TYPES IN STRUCTURED QUERY LANGUAGE (SQL)

Each column, local variable, expression, and parameter in SQL Server have a corresponding data type. The kind of data that an object can store, such

as binary strings, dates and times, characters, money, dates, and integers, as specified by the data type attribute of the object. SQL Server comes with a collection of system data types that describe all of the data types that can be used with it.

For each database, data types are divided into three categories (Figure 1.4):

- Types of String Data.
- Types of Numeric Data.
- Time and date Types of data.

Exact Numeric Data Types

They are used to hold precise numbers such as integers, decimals, and monetary amounts. The bit can store one of three values: 0 (zero), 1 (one), or NULL (null). Integer data is stored in the int, bigint, smallint, and tinyint data types. Numbers with fixed accuracy and scale are stored in decimal and numeric data types. It's worth noting that the terms decimal and numeric are interchangeable. Currency values are stored in the money and small money data types.

The features of the exact numeric data types are shown in the following table:

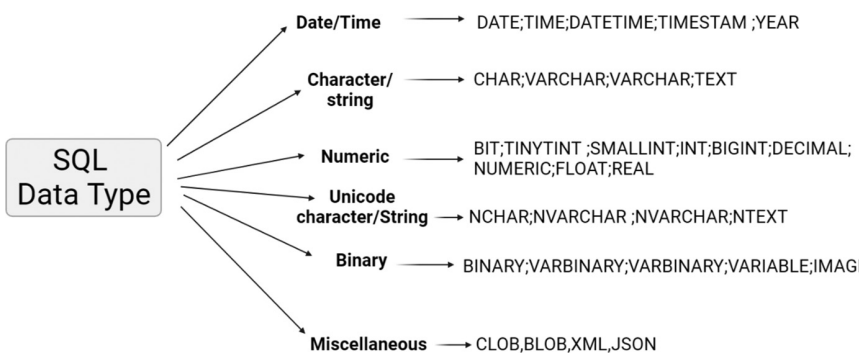


FIGURE 1.4 Data types of SQL.

Data Type	Lower Limit	Upper Limit	Memory
Big int	–2^63 (–9,223,372, 036,854,775,808)	2^63–1 (–9,223,362, 036,854,775,807)	8 bytes
int	–2^31 (–2,147, 483,648)	2^31–1 (–2,147, 483,647)	4 bytes
smallint	–2^15 (–32,767)	2^15 (–32,768)	2 bytes
tinyint	0	255	1 byte
bit	0	1	1 byte/8 bit column
decimal	–10^38+1	10^381–1	5–17 bytes
numeric	–10^38+1	10^381–1	5–17 bytes
money	–922,337, 203, 685,477.5808	+922,337, 203, 685,477.5807	8 bytes
Small money	–214,478.3648	+214,478.3647	4 bytes

Types of Numeric Data

Floating point numeric data is stored in the approximation numeric data type. In scientific computations, they are frequently utilised.

Data Type	Lower Limit	Upper Limit	Memory	Precision
float(n)	–1.79E + 308	1.79E + 308	Depends on the value of n	7 Digit
real	–3.40E + 38	3.40E + 38	4 bytes	15 Digit

Types of Date and Time Data

Date and time data, as well as the date time offset, are stored in the date and time data types.

Data Type	Storage Size	Accuracy	Lower Range	Upper Range
Date time	8 bytes	Rounded to increments of .000, .003, .007	1753-01-01	9999-12-31
Small date time	4 bytes, fixed	1 minute	1900-01-01	2079-06-06
date	3 bytes, fixed	1 day	0001-01-01	9999-12-31
time	5 bytes	100 nanoseconds	00:00:00.0000000	23:59:59.9999999
Date time offset	10 bytes	100 nanoseconds	0001-01-01	9999-12-31
datetime2	6 bytes	100 nanoseconds	0001-01-01	9999-12-31

When developing a new application, use the time, date, datetime2, and datetimeoffset data types. Because these types are more portable and conform to the SQL Standard. Furthermore, time, datetime2, and datetimeoffset offer greater precision in seconds, and datetimeoffset supports time zones.

Data Types for Character Strings

You can store either fixed-length (char) or variable-length (var) data in character strings data types (varchar). In the server's code page, the text data type can store non-Unicode data.

Data Type	Lower Limit	Upper Limit	Memory
char	0 chars	8000 chars	n bytes
varchar	0 chars	8000 chars	n bytes+2 bytes
varchar (max)	0 chars	2^31 chars	n bytes+2 bytes
text	0 chars	2,147,483,647 chars	n bytes+4 bytes

Character String Data Types in Unicode

Unicode character string data types store Unicode character data that is either fixed-length (nchar) or variable-length (nvarchar).

Data Type	Lower Limit	Upper Limit	Memory
nchar	0 chars	4,000 chars	2 times n bytes
nvarchar	0 chars	4,000 chars	2 times n bytes+2 bytes
ntext	0 chars	1,073,741,823 char	2 times the string length

Types of Binary String Data

The binary data types store binary data with fixed and variable lengths.

Data Type	Lower Limit	Upper Limit	Memory
binary	0 bytes	8,000 bytes	n bytes
varbinary	0 bytes	8,000 bytes	The length of data entered+2 bytes
image	0 bytes	2,147,483,647 bytes	

Other Data Types

Data Type	Description
cursor	For variables or stored procedure OUTPUT parameter that contains a reference to a cursor
row version	Open up a database to automatically created, distinct binary numbers.
A tree location in a tree hierarchy	It is represented by a hierarchyid.
unique identifier	16-byte GUID
sql_variant	Store values of other data types
XML	XML data should be stored in a column or XML-type variable.
spatial geometry	Use a flat coordinate system to represent data.
Type of spatial geography	Keep information that is ellipsoidal (round-earth), such as GPS latitude and longitude coordinates.
a table	temporarily stores a result set for processing later.

COMMANDS OF STRUCTURED QUERY LANGUAGE (SQL)

SQL is a data management solution that is free to use. The data from the table is retrieved and manipulated using a SQL query. It is used to run commands on a database or to create one. We can query, filter, sort, join, group, and alter data in a database using SQL commands. All of these commands in SQL Server are divided into four categories: DML, DDL, DCL, and TCL commands (Figure 1.5).

Data Definition Language (DDL)

The SQL instructions that can be used to define the database schema are known as DDL, or Data Definition Language. It only manages descriptions

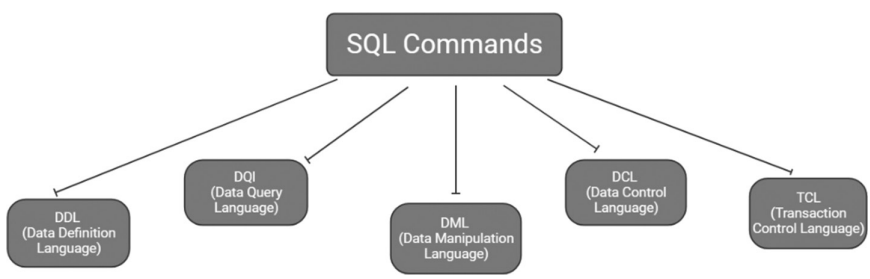


FIGURE 1.5 SQL commands.

of database schemas and is used to build and alter the structure of database objects in the database. In contrast to data, DDL refers to a set of SQL instructions for adding, removing, and altering database structures. These steps should not be carried out by a regular user who should be using an application to access the database.

The DDL commands are listed below:

- **Create:** The database and its objects are created by this command (like table, index, function, views, store procedure, and triggers).

  In SQL, you can use one of two CREATE statements:

  - CREATE A TABLE.
  - CREATE A DATABASE.

Make a Database

A database is a planned gathering of data. To store data in a well-structured manner, the first step with SQL is to establish a database. To build a new database in SQL, use the CREATE DATABASE statement.

Syntax:

```
CREATE DATABASE data_name;
```

Example:
It creates a new database in SQL and name the database as m database.

```
CREATE DATABASE m_database;
```

Make a Table

We have already learned how to create databases. To save the information, we'll need a table. In SQL, the CREATE TABLE statement is used to make a table. A table is made of rows and columns, as we all know. As a result, when building tables, we must give SQL with all relevant information, such as the names of the columns, the type of data to be kept in the columns, the data size, and so on. Let us look at how to utilise the Construct TABLE statement to create tables in SQL in more detail.

Syntax:

```
CREATE TABLE t_name
(
col1 data_type (size),
col2 data_type (size),
col3 data_type (size),
);
```

where t_name: name of the table; col1, col2, col3: names of the first, second and third columns of the table, and data_type: stores the type of data types we want to store in the particular column.

For example, the data type declared is int for integer data; size: defines the size we can store in a particular column like we specify the data_type as int and size as 15 then this column can store an integer with a maximum capacity limit of 15 digits.

This query will construct a table called Stud_Info, which will have three columns: R_NO, NAME, and SUBJECT.

```
CREATE TABLE Stud_info
(
R_NO int (5),
NAME varchar(35),
SUBJECT varchar(35),
);
```

A table called Stud_Info will be created as a result of this query. The R_NO field is of type int and can hold a five-digit integer value. The following two columns, NAME and SUBJECT, are of type varchar and can store characters, with the size 35 indicating that these two fields can hold a maximum of 35 characters.

- **Drop:** It is used to destroy database objects (whole database or simply a table). The DROP statement deletes existing objects such as databases, tables, indexes, and views. In SQL, a DROP command deactivates a component in a relational database management system (RDBMS).

  Syntax:

```
DROP object object_name
```

Examples:

```
DROP TABLE t_name; //t_name: Name of the table to be
deleted.
DROP DATABASE data_name; //data_name: Name of the
database to be deleted.
```

- **Alter:** It is used to change the database's structure. ALTER TABLE is used to modify an existing table by adding, deleting, or dropping columns. It can be used to create and remove constraints from a table that already exists.

Add to Table

To add columns to an existing table, use ADD. We may occasionally need to add extra information; in this instance, we do not need to recreate the entire database; instead, ADD comes to our rescue.

Syntax:

```
ALTER TABLE t_name
ADD (Col name_1 datatype,
     Col name_2 datatype,
     Col name_n datatype);
```

Drop a Table

The DROP COLUMN command is used to remove a column from a table. The table's unnecessary columns are removed.

Syntax:

```
ALTER TABLE t_name
DROP COLUMN col_name;
```

- **Truncate:** It is used to delete all records from a table, as well as all spaces reserved for the records.
  Syntax:

```
TRUNCATE TABLE t_name;
```

Example:

```
TRUNCATE TABLE Stud_Info;
```

- **Comment:** This will put your remarks in the data dictionary.

Syntax:

```
--(notes, examples)
```

Example:

```
--select the student data
SELECT * FROM Stud_Info;
```

- **Rename:** This is used to rename an existing database object.

Syntax:

```
RENAME old_ob_type ob name TO new_ob_name;
```

To give a proper explanation for above code, if we hadn't dropped the 'customers old data' table, we can have renamed it 'customer new data' as follows:

```
RENAME TABLE customers TO customer old_data;
```

Data Manipulation Language (DML)

The Data Manipulation Language is a set of data manipulation operators. We can alter data in the database by using these operators to add, modify, delete, or unload data.

The frequent SQL language operators are included in the group:

Select: collects data samples;

To begin, use the SELECT command to retrieve data from the database objects such as tables:

```
SELECT * FROM Stud_Info;
```

Insert: inserts new information;

this will help to insert data into tables. While working with the table, you can add additional records or rows. The sentence

is done by the keywords INTO and VALUES. Consider the case below:

```
INSERT INTO Student_Info (joining date , year of
Birth) VALUES
(2014-2-21, '1995');
```

Update modifies existing info.

We can select & insert data with SQL, but we can change it. To change current existing data in your tables, use the UPDATE command. It has a unique that is best described with an example.

We can replace the old inserted date of joining 2014-2-21 and year of birth as 1995-to something different using the following code. In our instance, that would be September 12, 2014:

```
UPDATE Stud_Info
SET Date of joining = '2014-09-12'
WHERE year of Birth = 1995;
```

Delete is a command that deletes data.

The DELETE command is identical to the TRUNCATE, with one notable exception. DELETE allows us to delete only the records you want from a table, as opposed to TRUNCATE, which allows us to delete every record in a table.

For instance, the below line of below will remove every record from the "sales" table:

```
DELETE FROM Student_Info;
```

Data Control Language (DCL)

The data control language is a SQL syntax that lets you govern user access to a database with just a couple of instructions. If database administrators have complete access to a database, they can also manage user access.

There are two DCL Commands:

Grant: GRANT gives users access to particular features. To run the command, use the following syntax:

```
DELETE FROM sales;GRANT type_of_permission ON
database_name.table_name TO '@username'@'localhost'
```

The line of code can be used to grant a certain kind of permission, such as complete or restricted access to the resources in a particular data table.

Revoke: The REVOKE clause, which is the opposite of the GRANT statement, is used to revoke database user permissions and privileges. On the other hand, their syntax is identical:

```
REVOKE type_of_permission ON database_name.table_name
FROM 'username'@'localhost'
```

In other words, instead of granting someone permission, you can revoke their privilege.

Transition Control Language (TCL)

When working with relational database management systems in a professional setting, it's vital to maintain control over your transactions. To put it another way, anytime you insert, delete, or change data in your database.

To deal with it, you must be conversant with TCL commands, specifically:

Commit: The changes you've made will be preserved in the database indefinitely, and other users will be able to access the updated version.

Assume you want to change the second client's last name from Amit to Astitiva in a record in the "Customers" table:

```
UPDATE customers
SET last_name = 'Astitiva'
WHERE customer_id = 2;
```

The task is not yet complete, though, as the other users of the database system won't be able to detect if we have made any modifications. To complete the operation, add a COMMIT statement at the end of the UPDATE block:

```
UPDATE customers
SET last_name = 'Astitiva'
WHERE customer_id = 2
COMMIT;
```

The transaction is saved to the database when you use COMMIT. The modifications are irreversible.

Rollback: The number of countries that are dedicated to the cause may quickly increase. For example, if you're the administrator, you might need to use COMMIT 20 times daily.

As a result, you may unintentionally insert or change data. The transaction control language function ROLLBACK allows you to undo any changes you don't want to keep permanently by restoring the database to its original committed state.

Add the ROLLBACK; to the end of your code to use this feature:

```
UPDATE customers
SET last_name = 'Astitiva'
WHERE customer_id = 2
COMMIT;
ROLLBACK;
```

Savepoint: A SAVEPOINT is a logical rollback point within a transaction. If you make a save point and then get an error, you can utilise the rollback option to undo everything you've done up to that point.

```
SAVEPOINT savepoint_name;
```

TRANSACTIONS IN SQL

It is a logical unit of work performed on a database. Transactions are logically ordered units or sequences of work that can be completed manually by a human or automatically by a database application. A transaction is the transmission of one or more changes to the database. An example of a table transaction would be the creation, update, or deletion of a record from that table. It is essential to maintain track of these transactions in order to protect data integrity and handle database concerns.

In practise, you'll group several SQL queries together and run them all at the same time as part of a transaction.

Characteristics of Transactional

Transactions contain the four standard qualities listed below, which are frequently abbreviated as ACID.

Atomicity ensures the successful completion of all processes inside the work unit. Otherwise, the transaction will be aborted at the point of failure, and all preceding activities would be reversed.

When a transaction is successfully committed, **consistency** ensures that the database changes states properly.

Isolation allows transactions to execute independently of one another while remaining transparent to them.

Durability ensures that a committed transaction's outcome or effect is preserved in the event of a system failure.

Commands for Controlling Transactions

The commands below are used to manage transactions:

- To save your changes, press Commit.
- TO REVERSE THE CHANGES, USE ROLLBACK.
- ROLLBACK POINTS are created by SAVEPOINT inside groups of transactions.
- SET TRANSACTION gives a transaction a name.

Commands for Transactional Control

DML commands like INSERT, UPDATE, and DELETE are the only ones that use transactional control commands. They can't be utilised when adding or removing tables because the database commits these activities automatically.

Commit (Command)

The transactional command COMMIT is used to save changes made by a transaction to the database. To save changes done by a transaction to the database, use the transactional verb COMMIT. The COMMIT command saves all transactions to the database that have occurred since the last COMMIT or ROLLBACK command.

The COMMIT command has the following syntax:

```
COMMIT;
```

Example:
Consider the following records in the CUSTOMERS table:

Sr. No.	ID	Name	Age	Address	Salary (USD)
1	LBR114	Alex	23	New York	25,000
2	LBR115	Joshua	21	Vegas	52,000
3	LBR116	Nick	22	Maryland	47,000
4	LBR117	Johnny	20	Boston	89,000
5	LBR118	Chris	28	Ohio	53,000
6	LBR119	Zyndaya	25	Omaha	48,000
7	LBR120	Tom	30	California	36,000

The example below will delete all records in the table with an age of 30 and then COMMIT the changes to the database.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 30;
SQL> COMMIT;
```

As a result, one row from the table will be removed, and the SELECT command yields the following result:

Sr. No.	ID	Name	Age	Address	Salary (USD)
1	LBR114	Alex	23	New York	25,000
2	LBR115	Joshua	21	Vegas	52,000
3	LBR116	Nick	22	Maryland	47,000
4	LBR117	Johnny	20	Boston	89,000
5	LBR118	Chris	28	Ohio	53,000
6	LBR119	Zyndaya	25	Omaha	48,000

Rollback (Command)
It is a command that allows you to go back in time. The transactional command is used to undo transactions that have not yet been recorded in the database. This command can only be used to reverse transactions that have occurred the last COMMIT or ROLLBACK command.

The ROLLBACK command has the following syntax:

```
ROLLBACK;
```

Example:
Consider the following records in the CUSTOMERS table:

Sr. No.	ID	Name	Age	Address	Salary (USD)
1	LBR114	Alex	23	New York	25,000
2	LBR115	Joshua	21	Vegas	52,000
3	LBR116	Nick	22	Maryland	47,000
4	LBR117	Johnny	20	Boston	89,000
5	LBR118	Chris	28	Ohio	53,000
6	LBR119	Zyndaya	25	Omaha	48,000
7	LBR120	Tom	30	California	36,000

The following is an example that would delete all records in the table with the age of 30 and then ROLLBACK the database modifications.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> ROLLBACK;
```

As a result, the delete action has no effect on the table, and the SELECT command yields the following result.

Sr. No.	ID	Name	Age	Address	Salary (USD)
1	LBR114	Alex	23	New York	25,000
2	LBR115	Joshua	21	Vegas	52,000
3	LBR116	Nick	22	Maryland	47,000
4	LBR117	Johnny	20	Boston	89,000
5	LBR118	Chris	28	Ohio	53,000
6	LBR119	Zyndaya	25	Omaha	48,000
7	LBR120	Tom	30	California	36,000

Savepoint (Command)
It allows you to save a point in time and let us revert to a previous state without reverting the entire transaction.

The SAVEPOINT command has the following syntax.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command is only used to create a SAVEPOINT in the middle of all transactional statements. To undo a collection of transactions, use the ROLLBACK command.

The following is the syntax for rolling back to a SAVEPOINT.

```
ROLLBACK TO SAVEPOINT_NAME;
```

The example below shows how to delete three separate records from the CUSTOMERS database. Before each delete, you should create a SAVEPOINT so that you can ROLLBACK to any SAVEPOINT at any moment to restore the required data to its previous state.

Example:
Consider the following records in the CUSTOMERS table.

Sr. No.	ID	Name	Age	Address	Salary (USD)
1	LBR114	Alex	23	New York	25,000
2	LBR115	Joshua	21	Vegas	52,000
3	LBR116	Nick	22	Maryland	47,000
4	LBR117	Johnny	20	Boston	89,000
5	LBR118	Chris	28	Ohio	53,000
6	LBR119	Zyndaya	25	Omaha	48,000
7	LBR120	Tom	30	California	36,000

The operations are listed in the following code block.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=LBR114;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID= LBR117;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=LBR120;
1 row deleted.
```

Now that the three deletions have occurred, let us pretend that you have changed your mind and have decided to ROLLBACK to the SP3 SAVEPOINT. The last two deletions are undone because SP2 was established after the initial deletion.

```
SQL> ROLLBACK TO SP3;
Rollback complete.
```

Since you rolled back to SP3, only the first deletion has occurred.

```
SQL> SELECT * FROM CUSTOMERS;
```

Sr. No.	ID	Name	Age	Address	Salary (USD)
1	LBR114	Alex	23	New York	25,000
2	LBR115	Joshua	21	Vegas	52,000
3	LBR116	Nick	22	Maryland	47,000
4	LBR117	Johnny	20	Boston	89,000
5	LBR118	Chris	28	Ohio	53,000
6	LBR119	Zyndaya	25	Omaha	48,000

Six Rows selected

Release Savepoint

It is a command that allows you to release a saved state.

The RELEASE SAVEPOINT command is used to delete a previously established SAVEPOINT.

The RELEASE SAVEPOINT command has the following syntax.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

We can no longer use the ROLLBACK command to undo transactions made since the last SAVEPOINT if a SAVEPOINT has been released.

Set Transaction

It is a command that allows you to set up a transaction. A database transaction can be started with the SET TRANSACTION command. This command is used to define the properties of the next transaction. You can, for example, make a transaction read-only or read-write.

The SET TRANSACTION command has the following syntax.

```
SET TRANSACTION [READ WRITE | READ-ONLY ];
```

VIEWS IN SQL

Views in SQL are comparable to virtual tables. A view's rows and columns are identical to those in a database table. By selecting fields from one or more database tables, a view can be made. Based on a criteria, a view may include all of a table's rows or only a subset of them.

In this article, we will discover how to build, remove, and update views.

Sn_ID	Name	Address
LRB123	Alex	New York
LRB124	Joshua	Vegas
LRB125	Nick	Maryland
LRB126	Johnny	Boston
LRB127	Chris	Ohio
LRB128	Zyndaya	Omaha

Student Details

Sn_ID	Name	Age	Marks
LRB123	Alex	15	91
LRB124	Joshua	16	95
LRB125	Nick	14	80
LRB126	Johnny	16	87
LRB127	Chris	14	98
LRB128	Zyndaya	15	99

Student Marks

To create a View, use the Build VIEW statement. A View can be built using a single table or several tables.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows
```

Examples:

Using a single table to create a View:

In this example, we'll use the table StudentDetails to create a View called DetailsView.

Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < LRB127;
```

To see the data in a view, we can query it similarly to how we query a table.

```
SELECT * FROM DetailsView;
```

Output:

Name	Address
Alex	New York
Joshua	Vegas
Nick	Maryland
Johnny	Boston

- In this example, we'll use the table StudentDetails to create a view called StudentNames.

Query:

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

If we now query the view as, we'll get the following results.

```
SELECT * FROM StudentNames;
```

Output:

Sn_ID	Name
LRB123	Alex
LRB124	Joshua
LRB125	Nick
LRB126	Johnny

- **Creating a View from Multiple Tables:** In this example, we'll use two tables, StudentDetails and StudentMarks, to create a View called MarksView. To create a View from many tables, we can just add additional tables to the SELECT statement.

Query:

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

View MarksView data should be displayed as follows:

```
SELECT * FROM MarksView;
```

Output:

Name	Age	Marks
Alex	15	91
Joshua	16	95
Nick	14	80
Johnny	16	87

Views Are Being Removed

We learned how to make views, but what happens if they are no longer needed? We will obviously wish to get rid of it. SQL can be used to delete an existing View. A View can be dropped or deleted via DROP statement.

Syntax:

```
DROP VIEW view_Name;
view_Name: Name of the View which we want to delete.
```

If we wish to delete the View MarksView, for example, we can do so as follows:

```
DROP VIEW MarksView;
```

Views Are Being Updated

- A view can only be updated if certain conditions are satisfied. If any one of the following requirements are not satisfied, we won't be able to update the view:
- Neither the GROUP BY clause nor the ORDER BY clause should be present in the SELECT statement used to generate the view.
- The DISTINCT keyword shouldn't be used in the SELECT statement.
- The View's values should all be NOT NULL.
- Complex or nested queries shouldn't be used to construct the view.
- A single table should be used to generate the view. If the view was created using multiple tables, it cannot be modified.

The CREATE OR REPLACE VIEW statement can be used to add or delete fields from a view.

Syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

For illustrate, if you wanted to modify the MarksView view and include the AGE field from the StudentMarks Table, we could accomplish it as follows:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS, StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

If we now retrieve all of the data from MarksView as:

```
SELECT * FROM MarksView;
```

Output:

Name	Address	Age	Marks
Alex	New York	15	91
Joshua	Vegas	16	95
Nick	Maryland	14	80
Johnny	Boston	16	87

- **Row Insertion in Views:** Row insertion in views works similarly to row insertion in tables. The SQL INSERT INTO statement can be used to add a row to a View.

  Syntax:

```
INSERT INTO view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);
view_name: Name of the View
```

  Example: In the example below, we will add a new row to the View DetailsView that we constructed before in the "building views from a single table" section.

```
INSERT INTO DetailsView (NAME, ADDRESS)
VALUES("Jaques", Texas");
```

  If we now retrieve all of the data from DetailsView as,

```
SELECT * FROM DetailsView;
```

Output:

Name	Address
Alex	New York
Joshua	Vegas
Nick	Maryland
Johnny	Boston
Jaques	Texas

Delete a Row from a View

Just like deleting rows from a table, deleting rows from a view is simple. Using the SQL DELETE statement, we can remove rows from a view. Additionally, after deleting a row from a table, deleting it from a view causes the change to appear in the view.

Syntax:

```
DELETE FROM view_Name
WHERE condition;
view_Name: Name of view from where we want to delete rows
condition: Condition to select rows
```

Example:
In this case, we'll delete the last row from the DetailsView view, which we just put in the previous row-inserting example.

```
DELETE FROM DetailsView
WHERE NAME="Jaques";
```

If we now retrieve all of the data from DetailsView as,

```
SELECT * FROM DetailsView;
```

Output:

Name	Address
Alex	New York
Joshua	Vegas
Nick	Maryland
Johnny	Boston

With Check Option

In SQL, the WITH CHECK OPTION clause is highly useful for views. It can be used with an updatable view. If the view is not updatable, inserting this clause in the CREATE VIEW statement is pointless. The WITH CHECK OPTION is used to prohibit rows from being inserted into the view of the condition in the CREATE VIEW statement's WHERE clause is not met. If the WITH CHECK OPTION clause was used in the CREATE

VIEW statement, and the UPDATE or INSERT clause does not meet the conditions, an error will be returned.

Example: Using the WITH CHECK OPTION clause, we create a View SampleView from the StudentDetails Table.

```
CREATE VIEW SampleView AS
SELECT S_ID, NAME
FROM  StudentDetails
WHERE NAME IS NOT NULL
WITH CHECK OPTION;
```

This view was constructed with the NOT NULL condition for the NAME column, so if we attempt to insert a new row with a null value in the Name column, we would see an error.

For example, despite the fact that the view is updatable, the following query for this View is invalid:

```
INSERT INTO SampleView(S_ID)
VALUES(8);
```

The NAME column's default value is null.

VIEWS AND THEIR APPLICATIONS

Views are necessary in a decent database for the reasons listed below:

- Data access is restricted.
- Views add another level of table protection by limiting access to a predetermined set of rows and columns in a database.
- Keeping data complexity hidden – a view can be used to mask the complexities of a multiple table join.
- Simplify user instructions—views allow users to choose data from several tables without having to know how to execute a join.
- Store complex queries in views—views can be used to keep track of complex queries.
- As the number of columns in the view matches the number of columns specified in the select statement, views can also be used to

rename columns without affecting the underlying tables. As a result, altering the base tables' columns can help hide the names of the columns.

- Ability to switch between multiple views for various users on the same table.

SQL COMMENTS

Comments can help you read and manage your application more easily. For example, you can add a comment to a statement that explains what the statement's purpose is in your application. Comments in SQL statements, with the exception of hints, have no effect on the statement's execution.

In a statement, a comment can exist between any keywords, parameters, or punctuation marks. There are two ways to include a comment in a statement:

Use a slash and an asterisk (/*) to start your comment. Continue with the comment's text. This text may be split across numerous lines. An asterisk and a slash (*/) should be used to close the comment. A space or a line break is not required to separate the opening and closing characters from the text. Begin your comment with the word – (two hyphens). Continue with the comment's text. This text isn't long enough to fill a new line. A line break should be used to end the comment. Additional limits apply to certain of the tools used to enter SQL. Multiple comments of both forms can be found in a SQL statement. The text of a remark may contain any printable characters from your database's character set.

The following three formats are available for making comments:

- **Comments on a single line (Single line comments)**
  Single line comments are those that start and end on a single line. A comment is a line that starts with '–' and will not be performed.
  Syntax:

```
-- single line comment
-- another comment
SELECT * FROM Customers;
```

- **Comments with multiple lines (Multi-line comments)**
  Comments that begin on one line and end on a different line are referred to as multi-line comments. Lines beginning with '/*'

are regarded to be the start of a comment and are deleted when '*/' appears.

Syntax:

```
/* multi-line comment
another comment */
SELECT * FROM Customers;
```

- **Comments in the margins (In line comments)**

  Comments can be mentioned in between statements and are enclosed in between '/*' and '*/'. An expansion of multi-line comments is n line comments; they can be stated in between statements and are enclosed in between '/*' and '*/'.

  Syntax:

```
SELECT * FROM /* Customers; */
```

Example:

There are numerous comments in these statements:

```
SELECT last_name, salary + NVL(commission_pct, 0),
   job_id, e.department_id
/* To select all employees whose compensation is
greater than that of Pataballa.*/
  FROM employees e, departments d
       /*The table DEPARTMENTS  is used to get the
department name.*/
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct,0) >   /*
Subquery:       */
   (SELECT salary + NVL(commission_pct,0)
 /* To total compensation is salary + commission_pct */
      FROM employees
      WHERE last_name = 'Pataballa');

SELECT last_name,                    -- select the name
    salary + NVL(commission_pct, 0),-- total compensation
    job_id,                          -- job
    e.department_id                  -- and department
  FROM employees e,                 -- of all employees
       departments d
```

```
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct, 0) >  -- whose
compensation
                                           -- is
greater than
      (SELECT salary + NVL(commission_pct,0)  -- the
compensation
    FROM employees
    WHERE last_name = 'Pataballa')        -- of Pataballa.
;
```

CONSTRAINTS IN SQL

Constraints are the guidelines that the data columns in a table must adhere to. These are used to restrict what kinds of data can be added to tables. This guarantees the database's data's dependability and accuracy. At the level of the column or table, constraints can be used. The column level constraints only apply to one column, while the table level constraints are applied to the entire table.

In SQL, the following constraints are available:

The following restrictions are possible in SQL:

Not Null: According to this constraint, a null value cannot be kept in a column. That is, we won't be able to save null values in a column if it is marked as NOT NULL.

When coupled with a column, the UNIQUE constraint demands that each and every value within the column be distinct. In other words, a column's values cannot be repeated in any row.

A field known as a PRIMARY KEY allows for the identification of each row in a table. And a table field is designated as the main key using this constraint.

Each row in another table can be uniquely identified by a field known as a FOREIGN KEY. A field may also be designated as a foreign key using this restriction.

Check: This constraint assists in comparing a column's values to a list of requirements. In other words, it helps to ensure that the value in a column meets a set of requirements.

When no value is entered by the user, this constraint specifies a default value for the column.

How should constraints be specified?

It can be specified when a table is created using the CREATE TABLE statement. After a table has been built, restrictions can also be specified using the ALTER TABLE statement.

The syntax for creating restrictions with the CREATE TABLE statement at the time of table creation is as follows.

```
CREATE TABLE sample_table
(
col1 data_type(size) constraint_name,
col2 data_type(size) constraint_name,
col3 data_type(size) constraint_name,
....
);
sample table is the name of the upcoming table.
data type: The category of information that can be
kept in the field.
constraint name: The constraint's name. UNIQUE,
PRIMARY KEY, and so on are a few examples.
```

Let's examine each restriction in further detail.

- **Not Null:** This stipulation must be fulfilled. If a table field is marked as NOT NULL. After that, a null value will never be accepted in the field. In other words, if you don't enter a value for this column, you won't be able to add a new row to the database. For instance, the following query generates a table. Student with the fields ID and NAME set to NOT NULL. That is, whenever we want to insert a new row, we must specify values for these two fields.

```
CREATE TABLE Student
(
ID int(8) NOT NULL,
NAME varchar(12) NOT NULL,
ADDRESS varchar(25)
);
```

- **Unique:** Each table row can be uniquely identified with the help of this constraint. In other words, all rows should have the same value for a given column. We can have a lot of unique columns in a table.

The given following query, for example, creates a table Student with the field ID UNIQUE. To put it in another way, it is noted that no two students can have the same ID.

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20)
);
```

- **Primary Key:** A primary key is a field in a table that uniquely identifies each row. If a field in a table is specified as a primary key, all rows must have unique values for this field, and it cannot include NULL values. In other words, this combines the NOT NULL and UNIQUE restrictions.

  This key can only be one field in a table. The following query will create a table called Student with the field ID as the primary key.

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20),
PRIMARY KEY(ID)
);
```

- **Foreign Key**: It is a table field that uniquely identifies each row of a different table. That is, this field refers to a table's main key. This usually results in a connection between the tables.

  Consider the following two tables:

Orders

ID	O_NO	C_ID
1	22	2
2	33	4
3	44	3
4	55	1

Customers

ID	NAME	ADDRESS
1	NAME	NOIDA
2	MAHESH	GURGAON
3	SURESH	DELHI

The field C_ID in the Orders table is clearly the main key in the Customers table, i.e., it uniquely identifies each row in the Customers dataset. As a result, the Orders table has a Foreign Key.

Syntax:

```
CREATE TABLE Orders
(
O_ID int NOT NULL,
ORDER_NO int NOT NULL,
C_ID int,
PRIMARY KEY (O_ID),
FOREIGN KEY (C_ID)  REFERENCES Customers(C_ID)
)
```

- **Check:** We can provide a requirement for a field's fulfilment while entering data by using the CHECK constraint.

  For instance, the query below creates a table called Student with the column AGE and the condition (AGE >= 18). In that case, the user won't be allowed to add any records with an age of under 18 to the database.

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
AGE int NOT NULL CHECK (AGE >= 18)
);
```

- **Default:** The fields' default value is provided by this restriction. In other words, the default value will be set to these fields if the user does not specify one while adding new records to the database.

For instance, the following query will create a table called Student and set the column AGE's default value to 18.

```
CREATE TABLE Student
(ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
AGE int DEFAULT 18
);
```

SQL CREATING ROLE

A role is designed to make the security model's setup and maintenance easier. It is a designated set of related privileges that a user can be awarded. It's tough to assign or revoke rights to users when there are a lot of them in a database. As a result, if you define roles as follows.

You can automatically grant or revoke privileges to users by providing or removing privileges. We can either create our own roles or use the pre-defined system roles. The following are some of the privileges offered to system roles:

System Roles	Role Connect is Granted
Certain privileges	Create a table, view, synonym, sequence, or session, for example.
Resource	Create a trigger, a table, a procedure, a sequence, etc. The Resource role's main function is to limit access to database objects.
DBA	Every system right.

Making a Role and Assigning It

The role must first be created by the (Database Administrator) DBA. The DBA can then provide the role privileges and assign users to it.

Syntax:

```
CREATE ROLE manager;
Role created.
```

The syntax for the role that has to be formed is called 'manager'.

- Now that the role has been formed, the DBA can assign users to it and assign privileges to it using the GRANT statement.

- In comparison to granting a permission to each user individually, it is simpler to GRANT or REVOKE privileges to users through a role.
- If a role is recognised by a password, the password must also be used to identify GRANT and REVOKE privileges.

Give a Role Privileges:

```
GRANT create table, create view
TO manager;
Grant succeeded.
```

Users Should Be Assigned a Role:

```
GRANT manager TO Johnny, Alex;
Grant succeeded.
```

Removing a Role's Privileges:

```
REVOKE create table FROM manager;
```

Drop a Role:

```
DROP ROLE manager;
```

Explanation to the above Syntax:
It first establishes a manager role, after which it permits managers to create tables and views. The role of manager is subsequently given to Johnny and Alex. Johnny and Alex are now able to construct tables and views. If a user has numerous roles assigned to them, they will have access to all of the roles' privileges. Then, using Revoke, the create table privilege is withdrawn from the role 'manager'. Drop is used to remove the role from the database.

SQL INDEXES

A SQL index is a type of index that allows you to quickly access data from a database. Without a question, one of the best methods to improve query and application performance is to index a database or view. A SQL index is a rapid lookup table used to find frequently sought records. An index is a data structure that is tiny, fast, and optimised for speedy lookups.

It's great for linking relational tables and searching huge databases. In SQL Server, indexes are utilised to speed up the query process, resulting in excellent performance. They're a lot like textbook indexes. If you need to get to a certain chapter in a textbook, you go to the index, identify the chapter's page number, and go straight to that page. Finding your selected chapter would have been extremely time consuming without indexes.

The same can be said for database indexes. Without indexes, a database management system (DBMS) must go through all of the records in a table to extract the needed results. This is known as table-scanning, and it is a very slow procedure. When you create indexes, on the other hand, the database looks for the index first and then obtains the table records directly.

Syntax for creating Index

```
CREATE INDEX index
ON TABLE column;
```

where the index is given a name, TABLE is the name of the table on which the index is constructed, and column is the name of the column to which it is applied.

For multiple columns:
Syntax:

```
CREATE INDEX index
ON TABLE (column1, column2,.....);
```

Unique Indexes

Unique indexes are used to maintain the integrity of the data in the table as well as to improve performance by preventing multiple entries from being entered into the table.

Syntax:

```
CREATE UNIQUE INDEX index
 ON TABLE column;
```

There are two types of indexes:

- Clustered index (index with clusters)
- Non-clustered index (index that isn't clustered)

Clustered Index

A clustered index specifies the physical order in which data is stored in a table. Each table can only include one clustered index since there is only one manner in which table data can be sorted. The primary key constraint in SQL Server automatically constructs a clustered index on that column.

Non-Clustered Indexes

The physical data inside the table is not sorted by a non-clustered index. In reality, a non-clustered index is kept in one location while table data is kept in another. This is similar to how a textbook is organised, with the information in one location and the index in another. This allows each table to have multiple non-clustered indexes.

It's worth noting that the data inside the table will be sorted using a clustered index. However, data is saved in the desired order within the non-clustered index. The index contains the index's column values as well as the address of the record to which the column value belongs.

When Should You Construct Indexes?

- A column can have a broad variety of values in it.
- A column does not have a lot of null values in it.
- In a when clause or a join condition, one or more columns are typically used together.

When Indexes Should Be Avoided

There are certain conditions mentioned below that should be considered:

- The table is modest in size.
- The columns aren't frequently used as a query condition.
- The column is frequently updated.

DROP INDEX

This command, you can delete an index from the data dictionary.

Syntax:

```
DROP INDEX index;
```

You must be the index owner or have the DROP ANY INDEX privilege to drop an index.

ALTERING INDEX

Altering an Index entails rebuilding or restructuring the index of an existing table.

```
ALTER INDEX IndexName
ON TableName REBUILD;
```

CONFIRMING INDEXES

You can verify the uniqueness of the different indexes present in a table given by the user or the server.

Syntax:

```
select * from USER_INDEXES;
```

It will display all of the server's indexes, where you may also find your own tables.

RENAMING AN INDEX

You can rename any index in the database using the system stored function sp rename.

Syntax:

```
EXEC sp_rename
   index_name,
   new_index_name,
   N'INDEX';
```

SEQUENCES IN SQL

A sequence is a collection of integers, such as 1, 2, 3, and so on, that some database systems construct and support in order to instantly produce unique values. A sequence is a schema-bound user-defined object that generates a list of numeric values. Many databases employ sequences because many applications need that each row in a table include a unique value, and sequences give an easy way to do so. The sequence of numeric numbers is generated at defined intervals in ascending or descending order, and it can be adjusted to resume when the max value is exceeded.

Syntax:

```
CREATE SEQUENCE "sequence_name"
START WITH "initial_value"
INCREMENT BY "increment_value"
MINVALUE "minimum value"
MAXVALUE "maximum value"
CYCLE|NOCYCLE ;
sequence_name: Name of the sequence.
initial_value: starting value from where the sequence
starts.
Initial_value should be greater than or equal to
minimum value and less than equal to maximum value.
increment_value: The value by which sequence will
increment itself. Increment_value can be positive or
negative.

minimum_value:The Minimum value of the sequence.
maximum_value: The Maximum value of the sequence.
cycle: When the sequence reaches its set_limit it
starts from the beginning.
nocycle: An exception will be thrown if sequence
exceeds its max_value.
```

Example:

The sequence query that creates the sequence in ascending order is shown below.

Example 1:

```
CREATE SEQUENCE sequence_1
start with 1
increment by 1
min value 0
maxvalue 100
cycle;
```

The query above will result in the creation of a sequence named sequence 1. The sequence will begin at 1 and will be incremented by 1 until it reaches a maximum value of 100. After exceeding 100, the sequence will repeat itself from the beginning.

Example 2:
Create a series in descending order using the sequence query.

```
CREATE SEQUENCE sequence_2
start with 100
increment by -1
min value 1
maxvalue 100
cycle;
```

The sequence 2 will be created as a result of the query above. The sequence will begin at 100 and will be incremented by –1 with a minimum value of 1 and should be less than or equal to the maximum value.

Create a table called students with the columns id and name as an example of how to utilise sequence.

```
CREATE TABLE students
(
ID number(10),
NAME char(20)
);
```

Insert values into the table now.

```
INSERT into students VALUES(sequence_1.nextval, 'Johnny');
INSERT into students VALUES(sequence_1.nextval, 'Simon');
```

where sequence 1. Nextval will add ids to the id column in the sequence defined in sequence 1.

ID	Name
LB 1	Johnny
LB 2	Simon

Query Processing in SQL

Query processing entails translating high-level queries into low-level expressions that may be employed at the file system's physical level, query optimisation, and query execution to obtain the desired result (Figure 1.6).

The figure shows the first step as the transformation of the query into a standard form. A query is translated into SQL and into a relational algebraic expression. During this process, the Parser checks the syntax and verifies the relations and the attributes which are used in the query. The second step is Query Optimiser, in which helps in transformation of the query

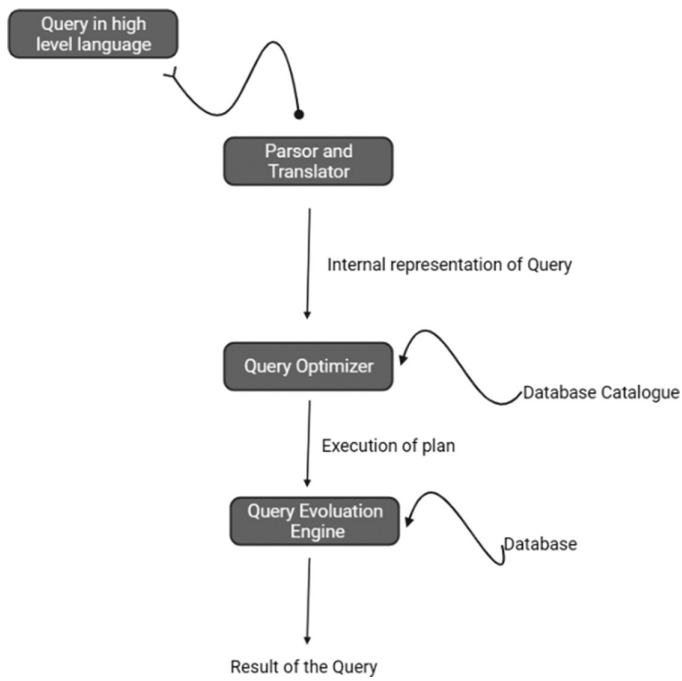


FIGURE 1.6 Query processing in SQL.

into equivalent expressions that are more efficient to execute. The third step is Query evaluation which will execute the above query execution plan and returns the result.

COMMON TABLE EXPRESSIONS (CTE) IN SQL

Common Table Expressions (CTE) were added into standard SQL to ease a variety of SQL queries for which a derived table was just not appropriate. It is a brief named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE query. It was introduced in SQL Server 2005. You may also utilise a CTE as part of the SELECT query in a CREATE a view. Furthermore, starting with SQL Server 2008, you can use a CTE with the new MERGE statement.

Defining CTEs

CTEs can be defined by including a WITH clause before any SELECT, INSERT, UPDATE, DELETE, or MERGE statement. One or more CTEs can be separated by commas in the WITH clause. You can use the following syntax:

```
[WITH  [, ...]]
 ::=
cte_name [(column_name [, ...])]
AS (cte_query)
```

You can then refer to the CTEs as you would any other table after you've defined your Using clause with the CTEs. You can only refer to a CTE within the execution scope of the statement that comes after the WITH clause. The result set is not available to other statements once you've run your statement.

Creating a Common Table Expression (Recursive)

A recursive CTE is one that refers back to itself inside itself. When working with hierarchical data, the recursive CTE is handy since it continues to execute until the query returns the whole hierarchy. A table with a list of employees is a good example of hierarchical data. The table includes a link to the manager's contact information for each employee. Within the same table, such reference is an employee ID. A recursive CTE can be

used to show the hierarchy of employee data. A CTE might become stuck in an infinite loop if it is built incorrectly. The MAXRECURSION hint can be appended to the OPTION clause of the primary SELECT, INSERT, UPDATE, DELETE, or MERGE statement to prevent this.

Types of Common Table Expressions

CTEs are divided into two categories: There are two types of recursive functions: recursive and non-recursive.

- **Non-Recursive CTEs**

  Non-Recursive CTEs are simple CTEs that do not employ recursion or repeat processing in a subroutine. We'll make a simple non-recursive CTE to show the number of rows from 1 to 10. Each and every CTE query will begin with a "With," followed by the CTE Expression name and column list, according to the CTE Syntax.

  Example: Create a table

```
CREATE TABLE Empl_Info
(
  Emp_ID int NOT NULL PRIMARY KEY,
  F_Name varchar(50) NOT NULL,
  L_Name varchar(50) NOT NULL,
  M_ID int NULL
)
INSERT INTO Employees VALUES (1, 'Kalvin', 'Thom', NULL)
INSERT INTO Employees VALUES (2, 'Troy', 'Bolton', 1)
INSERT INTO Employees VALUES (3, 'Rob', 'Durello', 1)
INSERT INTO Employees VALUES (4, 'Robby', 'Bailey', 2)
INSERT INTO Employees VALUES (5, 'Ken', 'Erickson', 2)
INSERT INTO Employees VALUES (6, 'Bill', 'Goldberg', 3)
INSERT INTO Employees VALUES (7, 'Robert', 'Miller', 3)
INSERT INTO Employees VALUES (8, 'Duke', 'Mark', 5)
INSERT INTO Employees VALUES (9, 'Chugs', 'Matthew', 6)
INSERT INTO Employees VALUES (10, 'Morson', 'Jhonson', 6)
```

  After the Empl_Info is created, following SELECT statement, which is preceded by a WITH clause that includes a CTE named cte Reports is created:

```
WITH
  cteReports (Emp_ID, F_Name, L_Name, M_ID, EmpLevel)
  AS
  (
    SELECT Emp_ID, F_Name, L_Name, M_ID, 1
    FROM Empl_Info
    WHERE M_ID IS NULL
    UNION ALL
   SELECT e.Emp_ID, e.F_Name, e.L_Name, e.M_ID,
      r.EmpLevel + 1
    FROM Empl_Info e
      INNER JOIN cteReports r
        ON e.M_ID = r.Emp_ID
  )
SELECT
  F_Name + ' ' + L_Name AS Full Name,
  EmpLevel,
  (SELECT F_Name + ' ' + L_Name FROM Empl_Info
    WHERE Emp_ID = cteReports.M_ID) AS Manager
FROM cteReports
ORDER BY EmpLevel, M_ID
```

When you need to build temporary result sets that can be retrieved in a SELECT, INSERT, UPDATE, DELETE, or MERGE statement, CTEs can be a handy tool.

TRIGGERS IN SQL

A trigger is a database stored procedure that is automatically invoked whenever a specific event happens in the database. A trigger can be triggered when a row is entered into a table or when particular table columns are modified, for example.

```
create trigger [trig_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

Syntax Explanation:

- **Make a Trigger [Trigger Name]:** The trigger name creates or replaces an existing trigger.
- **[Before | After]:** This determines the order in which the trigger will be run.
- **This Indicates the DML Operation:** insert | update | delete.
- **On [Table Name]:** This indicates the name of the table that the trigger is linked with.
- **[For Each Row]:** This is a row-level trigger, which means it will be executed for each row that is affected.
- **[Trigger Body]:** This specifies the action to be taken when the trigger is triggered.

Before and After Triggers

BEFORE triggers execute the trigger action prior to the execution of the triggering statement. AFTER triggers execute the trigger action following the execution of the triggering statement.

As an example, consider a Student Report Database in which student grades are kept. Create a trigger in such a scheme to automatically insert the total and average of specified marks whenever a record is inserted. BEFORE Tag can be used here since the trigger will fire before the record is inserted.

Assume you have a Database schema –

```
mysql> desc Student;
```

Field	Type	Null	Key	Default	Extra
tid	int(4)	NO	PRI	NULL	auto _increment
name	varchar(30)	YES		NULL	
subj1	int(2)	YES		NULL	
Subj2	int(2)	YES		NULL	
Subj3	int(2)	YES		NULL	
Total	int(3)	YES		NULL	
Per	int(3)	YES		NULL	

To the problem statement, add a SQL trigger

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 +
Student.subj3, Student.per = Student.total * 60 / 100;
```

The above SQL line will build a trigger in the student database so that anytime subjects marks are entered, the trigger will compute those two values and insert them with the entered values before entering the data into the database. i.e.

```
mysql> insert into Student values(0, "ABCDE", 20, 20,
20, 0, 0);
```

Query OK, 1 row affected (0.09 sec)

```
mysql> select * from Student;
```

Tid	Name	Subj1	Subj2	Subj3	Total	Per
100	ABCDE	20	20	20	60	36

Triggers can be created and executed in databases in this manner.

BOOK MANAGEMENT DATABASE IN SQL TRIGGER

Given a Library Book Management database schema and a Student database structure, for example. A student who takes out a book from the library, the count of that specific book should be decremented in these databases. In order to do so,

Assume the schema has some data:

```
mysql> select * from book_det;
```

Bid	Battle	Copies
1	MySql	10
2	C++	5
3	Java	15
4	Oracle DBMS	10

```
mysql> select * from book_issue;
```

Bid	sid	title

To create such a mechanism, a trigger should automatically invoke and decrease the copies attribute by 1 if the system enters data into the book issue database, allowing for accurate book tracking.

The system's trigger:

```
create trigger book_copies_deducts
after INSERT
on book_issue
for each row
update book_det set copies = copies -1 where bid =
new.bid;
```

When an insertion operation is conducted in a book issue database, the above trigger is triggered, and the book det schema setting copies decrements by 1 of the current book id (bid).

Results –

```
mysql> insert into book_issue values(1, 100, "Java");
```

Query OK, 1 row affected (0.09 sec).

```
mysql> select * from book_det;
```

Bid	Battle	Copies
1	MySql	10
2	C++	5
3	Java	15
4	Oracle DBMS	10

```
mysql> select * from book_issue;
```

Bid	Sid	Title
1	100	MySql

As shown in the findings above, as soon as data is inserted, copies of the book are deducted from the system's book schema.

INTRODUCTION TO NoSQL (NON-RELATIONAL SQL)

A NoSQL database, which stands for 'non-SQL' or 'non-relational', is a database that allows for data storage and retrieval. Other than the tabular relations utilised in relational databases, this data is modelled in a different way. Although these databases first debuted in the late 1960s, the term 'NoSQL' wasn't coined for them until the early 2000s. Large-scale data analytics and real-time internet applications increasingly use NoSQL databases. The phrase 'not only SQL' is used to highlight the possibility of SQL-like query languages being supported by NoSQL systems. An advantage of a NoSQL database is its ease of design, horizontal scaling to server clusters, and tighter control over availability. NoSQL databases employ different default data structures than relational databases, which enables NoSQL to carry out some operations more quickly. The problem that a NoSQL database is meant to solve determines how applicable it is. There is a misconception that NoSQL databases' data structures are more adaptable than relational databases' tables.

Many NoSQL databases make trade-offs between consistency and availability, performance, and partition tolerance. The usage of low-level query languages, a lack of standardised interfaces, and large prior investments in relational databases are all barriers to wider adoption of NoSQL storage. Although most NoSQL databases lack true ACID transactions (atomicity, consistency, isolation, and durability), a few databases, including MarkLogic, Aerospike, FairCom c-treeACE, Google Spanner (though technically a NewSQL database), Symas LMDB, and OrientDB, have incorporated them into the core of their designs.

The majority of NoSQL databases allow for eventual consistency, which allows for the gradual propagation of database modifications across all nodes. Stale readings, a problem caused by reading outdated data, might occur as a result of data searches that don't immediately return new results. Additionally, lost writes and other types of data loss may occur in some NoSQL systems. Some NoSQL systems provide solutions like write-ahead logging to prevent data loss. When doing distributed transaction processing over numerous databases, achieving data consistency is much more challenging. Both NoSQL and relational databases struggle with this. Even today's relational databases don't allow cross-database referential integrity constraints. Few systems support both the X/Open XA standards and ACID transactions for distributed transactions processing.

A BRIEF HISTORY OF NoSQL DATABASES

- 1998: Carlo Strozzi used the name 'NoSQL' to describe his light-weight, open-source relational database Neo4j is released in 2000.
- Google BigTable was launched in 2004.
- CouchDB was launched in 2005.
- The research paper on Amazon Dynamo is published in 2007.
- Facebook releases the Cassandra project to the public in 2008.
- The phrase NoSQL was resurrected in 2009.

Types of NoSQL Databases

NoSQL databases come in four different varieties: key-value pair, column-oriented, graph-based, and document-oriented. Each group has its own set of characteristics and limits. None of the databases listed above is better at solving all of the difficulties. Users should choose a database that meets their product requirements.

NoSQL databases are available in a range of forms and dimensions.

- Key-value
- Column-oriented based on pairs
- Graphs-based
- Document-oriented Graphs

KEY-VALUE

They are used to store data, and they are made to withstand tremendous loads and big amounts of data. Key-value pair storage databases employ hash tables to store data; each key must be unique, and the value may be in text, JSON, BLOBs (Binary Large Objects), or another format.

COLUMN-BASED

These are based on Google's BigTable paper and function with columns. Each column is dealt with independently. The values of single-column databases are kept together.

DOCUMENT-ORIENTED

The value part of a NoSQL DB's data is kept as a document, while the key-value pair is stored as a key-value pair. JSON or XML formats are used to store the document. The database recognises the value and can be queried. CMS systems, blogging platforms, real-time analytics, and e-commerce apps all use this document type. It should not be used for complex transactions involving several operations or queries against different aggregate models.

Popular Document DBMS systems include Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, and MongoDB.

GRAPH-BASED

This maintains both entities and the relationships between them. The entity is represented as a node, while the relationships are represented as edges. An edge establishes a connection between nodes. A unique identifier is assigned to each node and edge. It is multi-relational in nature, as opposed to a relational database, which has loosely connected tables. Traversing relationships is quick because they are already stored in the database and don't need to be calculated. Graph databases are commonly used for social networks, logistics, and geographic data.

Popular graph-based databases include Neo4J, Infinite Graph, OrientDB, and FlockDB.

FEATURES OF NoSQL

The following are the feature:

Non-Relational Database Management System (NoSQL) Features

- The relational model is never followed by NoSQL databases.
- Tables with flat fixed-column records should never be used.
- Work with BLOBs or self-contained aggregates.
- Data normalisation and object-relational mapping are not required.
- There are no advanced features such as query languages, query planners, referential integrity joins, or ACID compliance.

Schema-Free

- NoSQL databases are either schema-free or contain schemas that are more loose.
- There is no requirement for any kind of data schema specification.
- Provides data structures that are heterogeneous within the same domain.

API That Is Easy to Use

- Provides APIs that provide low-level data manipulation and selection methods, as well as easy-to-use interfaces for storing and accessing data.
- Protocols based on text; typically used in conjunction with HTTP REST and JSON.
- Web-enabled databases that run as internet-facing services mostly employ the NoSQL query language, which is not based on any standard.

Distributed

- A distributed execution of many NoSQL databases is possible.
- Auto-scaling and fail-over capabilities are included.
- The ACID principle is frequently overlooked in favour of scalability and throughput.
- Asynchronous replication across remote nodes is almost non-existent. HDFS Replication, Asynchronous Multi-Master Replication, Peer-to-Peer.
- Only ensuring long-term consistency.
- Nothing is shared in the architecture. As a result, there is less coordination and more dispersal.

NoSQL Query Mechanism Tools

The most common method of data retrieval is the REST-based GET resource, which retrieves a value based on its key or ID. Because they grasp the value in a key-value combination, document store databases allow for more challenging searches. With MapReduce, CouchDB, for example, allows you to define views.

WHAT IS THE CAP THEOREM, AND HOW DOES IT WORK?

Brewer's theorem is another name for the CAP theorem. It claims that a distributed data store cannot provide more than two out of three guarantees.

- Consistency
- Availability
- Tolerance for Partitions

Consistency

Even after an operation has been completed, the data should stay consistent. This indicates that once data is written, it should be included in any subsequent read requests. After altering the order status, for example, all clients should be able to see the same information.

Availability

The database should be accessible and responsive at all times. There should be no downtime.

Tolerance for Partitions

Partition Tolerance means that the system should keep working even if the connection between the servers isn't always reliable. The servers, for example, can be divided into several groups that may or may not communicate with one another. If one portion of the database is down, the other sections are always up and running.

CONSISTENCY IN THE LONG RUN

To achieve high availability and scalability, 'eventual consistency' refers to having copies of data on several machines. Any modifications made to a data item on one system must be duplicated on all other replicas as a result. Data replication may not be instantaneous since some copies will be updated right once while others will take longer. These copies may be

inconsistent at first, but with time, they become consistent. As a result, the term 'ultimate consistency' was coined.

BASE stands for Basic Availability, Soft State, and Eventual Consistency.

According to the CAP theorem, availability indicates that the database is available at all times.

The term 'soft state' refers to a system's ability to alter even when no input is provided.

The term 'eventual consistency' refers to the system's ability to become consistent over time.

WHEN SHOULD YOU UTILISE NoSQL?

- When you need to store and retrieve a large volume of data.
- The relationship between the data you keep isn't as significant as you might think.
- The information is fragmented and dynamic.
- At the database level, support for constraints and joins is not necessary.
- The data is always growing, and you'll need to scale the database on a regular basis to keep up with it.

ADVANTAGES

Working with NoSQL databases such as MongoDB and Cassandra has numerous advantages like high scalability and availability:

- **High Scalability:** NoSQL databases use sharding to provide horizontal scale. Data division and distribution across several machines while preserving the data's order is known as sharding. In order to manage the data, new machines must be added horizontally, and existing machines must have more resources added vertically. While horizontal scaling is simple to perform, vertical scaling is more challenging. Examples of databases with horizontal scaling include MongoDB, Cassandra, and others. NoSQL can handle a lot of data since it is scalable. As the data expands, NoSQL scales to handle effectively and efficiently.
- **High Availability:** The auto-replication functionality in NoSQL databases makes them highly available because data replicates itself to a previous consistent state in the event of a failure.

DISADVANTAGES

NoSQL has the following drawbacks:

- **Narrow Focus:** NoSQL databases have a very restricted focus because they are primarily built for storage and provide very little functionality. In the topic of Transaction Management, relational databases outperform NoSQL databases.
- **Open Source:** NoSQL is a database that is free to use. There is currently no reliable NoSQL standard. Thus, there is a considerable likelihood that two database systems will differ from one another.
- **Management Challenge:** The goal of big data tools is to make managing vast amounts of data as simple as feasible. But it isn't that simple. Data administration in a NoSQL is far more complicated than it is in a relational database. It is known for being difficult to set up and far more difficult to administer on a daily basis.
- **GUI Mode Not Available:** GUI mode tools to access the database are not flexible in the market.
- **Backup:** For some NoSQL databases, such as MongoDB, backup is a major flaw. MongoDB does not include a method for backing up data in a consistent manner.
- **Large Document Size:** Data is stored in JSON format in some database systems, such as MongoDB and CouchDB. This means that documents are quite huge (due to BigData, network bandwidth, and speed), and having detailed key names actually harms the document size by increasing it.

SUMMARY

In this chapter, we have covered all the fundamentals of SQL, including commands, statements, data types, views, comments that create roles, indexes, query processing, CTE, database management, and a quick introduction to NoSQL.

NOTE

1 Codd, E.F., 1983. A relational model of data for shared data banks. *Communications of the ACM*, 26(1), pp. 64–69.

CHAPTER 2

Clauses/Operators

IN THIS CHAPTER

- Basics clauses/operators of SQL
- Features of various clauses/operators in SQL
- Syntax and examples

An operator is a reserved word or character that is used in the WHERE clause of a SQL statement to conduct operations like comparisons and arithmetic computations. These operators are used to express conditions in SQL statements and to act as conjunctions for numerous conditions in a single query.

- Operators in arithmetic
- Operators for comparison
- Operators logical
- Negative operators were employed to negate conditions

WITH CLAUSE IN SQL

Oracle introduced the WITH clause in the Oracle 9i release two databases. The SQL WITH clause enables you to give a sub query block a name

DOI: 10.1201/9781003358435-2

(a process known as subquery refactoring) that can be referred from various locations within the SQL query.

- The clause is used to define a temp relation so that the output of this temporary relation is available and can be used by the query connected with it.
- Queries with a linked WITH clause can be created with nested subqueries, which improves the readability and debug ability of the SQL query.
- All database systems don't support the WITH clause.
- The name given to the sub query is treated as if it were a table or inline view.
- Oracle introduced the WITH clause in the Oracle release 2 database.

Syntax:

```
WITH temporaryTable (averageValue) as
    (SELECT avg(Attr1)
    FROM Table)
    SELECT Attr1
    FROM Table, temporary Table
    WHERE Table.Attr1 > temporary Table.average value;
```

The WITH clause is used to create a temp table with only one attribute, average Value. The average value of column Attr_1 in relation Table is stored in average Value. Following the WITH clause, the SELECT statement will return those tuples in which the value of Attr1 in relation Table is greater than the avg value acquired from the that clause statement.

When a query with a WITH clause is run, the clause query is evaluated first, and the result of that evaluation is saved in a temporary relation. The primary query linked with the WITH clause is then eventually executed, that uses the temporary relation created.

Queries:

Example 1: Here, we identify all employees whose salaries are higher than the average of all employees.

Relationship name: Employee

E_ID	Name	Salary
11	Steve	45,000
22	Johnny	95,000
27	Simon	78,550
45	Waltz	60,000
85	Sheldon	68,000
97	Katherine	99,000

Query in SQL:

```
WITH temporary Table(averageValue) as
    (SELECT avg(Salary)
    from Employee)
        SELECT EmployeeID,Name, Salary
        FROM Employee, temporaryTable
        WHERE Employee.Salary > temporaryTable.
        averageValue;
```

Explanation: The avg salary of all employees is 74,258. As a result, all employees whose salaries are higher than the average are included in the output relationship.

E_ID	Name	Salary
22	Johnny	95,000
27	Simon	78,550
97	Katherine	99,000

Output:

Example 2: Locate those airlines in which the whole salary of all pilots in that airline exceeds the database's average total salary of all pilots. Pilot is the name of the relationship.

E_ID	Airline	Name	Salary (Rupees)
40007	Air India	Kumar	80,000
40002	Jet Airways	Lawrence	50,000
20027	Air India	Watson	40,050
60778	Indigo	Crick	80,780
85585	Jet Airways	Steve	25,000
94070	Air India	Kim	78,000

Explanation: The overall income of all Air India pilots is 66,016.6, while the total salary of all Jet Airways pilots is 37,500. In the table Pilot, the average income of all pilots is 58,971.6. Because only the total income of all Air India pilots is more than the average salary, Air India is in the output relationship.

Output:

Airline
Air India

When dealing with sophisticated SQL statements rather than basic ones, the SQL WITH clause comes in handy. It also allows you to split down large SQL queries into smaller chunks, making debugging and processing the queries much easier. The WITH clause in SQL is essentially a drop-in replacement for the subquery.

WITH TIES CLAUSE IN SQL

WITH TIES enables you to return more rows with values matching the last row in the limited result set. It is important to note that WITH TIES may result in more rows being returned than you specified in the expression. The clause WITH TIES can only be used in conjunction with TOP and ORDER BY; both clauses are necessary.

The SQL Server SELECT TOP command retrieves records from one or more SQL Server tables while limiting the number of entries returned based on a specified value or percentage.

Syntax:

```
SELECT TOP (top_value) [ PERCENT ] [ WITH TIES ]
expressions
FROM tables
[WHERE conditions]
[ORDER BY expression [ ASC | DESC ]];
```

ARITHMETIC OPERATORS IN SQL

The arithmetic operators in SQL are used to perform mathematical operations on data stored in database tables, such as addition, subtraction, multiplication, division, and modulus, among others. These arithmetic

operators can be used with a WHERE clause in a SQL statement if there are multiple conditions in the query that need to be satisfied, but if there is any Null value present in the table, performing arithmetic operations on the Null value will result in an error.

The Top five Arithmetic Operators in SQL are:

Addition (+), subtraction (–), multiplication (*), division (/), and modulus (percent) are the various arithmetic operators in SQL that are used to conduct mathematical operations on data stored in database tables. Let's look at the examples below to see how the various arithmetic operators in SQL function.

Addition Operator (+)

The addition of something is denoted by the plus sign (+). This operator '+' is used to add two digits together. We can see how the addition operator is used to add 110 and 320 in the example below.

```
SELECT 110 + 320 as Addition;
```

Output:

Addition
430

Example: To illustrate how the operators function, consider the table 'EMPLOYEES' shown below:

ID	Name	Age	Salary
100	SARESH	27	30,000
150	SUNIL	21	45,000
185	RISHABH	22	50,000
200	SHASHANK	25	35,000

The table lists numerous employees and their contact information. Let's combine the two columns together in the SALARY column. The SALARY column is increased by 10,000 in the query below.

```
SELECT SALARY+10000 as new_salary FROM EMPLOYEES;
```

Output:

N_Salary
40,000
55,000
60,000
45,000

Let's add two columns together using the addition operator, as indicated in the query below.

```
SELECT SALARY+ID as add_salary FROM EMPLOYEES;
```

The above query has following results, along with the salary and ID columns.

Output:

Add_Salary
40,100
55,150
60,185
452,000

Subtraction Operator (–)
Using the subtraction operator '–,' the right-hand operand is subtracted from the left-hand operand. Let us see at the example below to see how to subtract 100 from 360.

```
SELECT 360-100 as Subtract;
```

Subtract
260

Take, for example, the table 'EMPLOYEES' that was previously mentioned. We can see that 500 is deducted from the SALARY column in the query below.

```
SELECT SALARY-500 as S_Sal FROM EMPLOYEES;
```

Output:

S_Sal
29,500
44,500
49,500
34,500

The subtraction of two columns, i.e., in the query below, is performed. The following information is displayed: SALARY and ID.

```
SELECT SALARY-ID as N_Sal FROM EMPLOYEES;
```

The ID column is deducted from the SALARY columns in the above query, and the result is as follows:

N_Sal
29,400
44,350
49,315
34,300

Operator for Multiplication (*)

The multiplication of two operands is performed by this operator. The multiplication of 10 and 80 can be seen in the example below.

```
SELECT 10*80 as Multiplication;
```

Output:

Multiplication

800

Using the EMPLOYEES table as a starting point, multiply the column SALARY by 15 as indicated below:

```
SELECT SALARY*10 as Multi_Salary FROM EMPLOYEES;
```

The above query has following results, and we can see that the SALARY column is multiplied by 10.

Multi_Salary
300,000
450,000
500,000
350,000

The following query shows the multiplication of two columns from the table EMPLOYEES, namely, SALARY and ID.

```
SELECT SALARY*ID as A_Salary FROM EMPLOYEES;
```

Output:

A_Salary
300,000
6,750,000
9,250,000
7,000,000

Division Operator (/)

The left-hand side operand is divided by the right-hand side operand with this operator. The division process is performed in the example below, where 20 is divided by 5.

```
SELECT 20/5 as Division;
```

The output of the aforementioned operation is the quotient of the division, which is four in this case.

Output:
Division
4

Consider the table 'EMPLOYEES', which we previously discussed. The query below shows how the column SALARY is divided by 5 in the division operation.

```
SELECT SALARY/5 as Sal_Div FROM EMPLOYEES;
```

Sal_Div
6,000
9,000
10,000
7,000

Modulus Operator(%)
The residual of the division of the left-hand side operand by the right-hand side operand is obtained using this arithmetic operator.

```
SELECT 12%4 as result;
```

The preceding question indicates that when 12 is divided by 4, the remainder of the division, 0 is returned as the output, as shown in the result.

Result:
0

The SALARY column of the EMPLOYEES table below is used to perform the modulus operation.

```
SELECT SALARY%100 as result FROM EMPLOYEES;
```

The above query conducts the modulus operation, and when the employees' salaries are divided by 100, the result shows the division's remaining values, is mentioned as output.

Salary
300
450
500
350

If we execute any arithmetic operation on NULL, the result will always be null. The SQL arithmetic operators are essential for executing complex mathematical calculations on data in database tables. It is critical for developers to have a strong grasp of these operators.

WILDCARD IN SQL: AN OVERVIEW

In SQL, a wildcard character replaces zero to any number of characters in a string. These wildcard characters are usually found in conjunction with the SQL operator LIKE. This is a character search operator that is widely used in SQL's WHERE clause to find a specific set of characters. Regular Expressions and Wildcards both have the same aim. This character that can be substituted for any other character or characters in a string. These wildcards come in handy when we need to conduct a rapid search in the database.

In SQL, there are two frequent wildcard characters. The % sign can be used to represent zero, one, or any number of characters. The underscore symbol designates a single character, which can be either a letter or a number. There are numerous ways to arrange these symbols.

We'll take a closer look at these wildcard characters, as well as a few additional essential wildcard characters.

Wildcard	Description	Example
% (Percent sign)	Zero or more character	bl% finds bl, black, blue, and blob
_ (Underscore)	Only one character	p, t discover pot, pit, and put, is matched.
[] (Square brackets)	matches a single character supplied within the brackets	The wildcard p[oi]t find pot and pit, but not put,
^ (Caret)	This wildcard matches characters that appear outside brackets	p[oi]t finds put but not pot and pit

To match zero or more characters, Microsoft Access utilises the character asterisk symbol (*) instead of the percent symbol (percent) wildcard character, and it uses the character question mark (?) instead of the underscore wildcard character to indicate a single character.

SQL Wildcards Syntax

Let's see how the wildcard characters 'percent' and '_' might be written in several ways:

```
SELECT FROM table_name
WHERE column LIKE 'BB%'
```

BB% allows us to search for strings that start with BB and end with another single character or multiple characters.

Or

```
SELECT FROM table_name
WHERE column LIKE '%BB%'
```

We can use %BB% percent to find strings that start with any number of characters but contain the string BB in the middle and end with any number of characters between 0 and infinity.

Or

```
SELECT FROM table_name
WHERE column LIKE '_BB'
```

_BB allows us to search for strings that start with a single distinct character and end with the character BB.

Or

```
SELECT FROM table_name
WHERE column LIKE 'BB_'
```

We can use BB_ to find any strings that start with the BB pattern and terminate with a single different character.

Or

```
SELECT FROM table_name
WHERE column LIKE '_ BB _'
```

_BB_ allows us to search for strings that start with a unique character, contain the pattern BB in the middle, and end with a single unique character.

Regular Expressions and wildcard characters both perform the same thing. To improve search outcomes and results, we can mix numerous wildcards in a single string. A distinct SQL wildcard could be used for a similar function in a few databases, such as MS Access.

EXCEPT AND INTERSECT OPERATORS

Let us learn about each operators individually as mentioned below.

Intersect Clause: It is used to convey the result of the intersection of two select statements, as the name implies. This means that the result will include all of the rows that are common to both SELECT commands.

Syntax:

```
SELECT column-1, column-2 ……
FROM table 1
WHERE…..
INTERSECT
SELECT column-1, column-2 ……
FROM table 2
WHERE…..
```

Consider the following scenario:

Employee Information (Table 1)

ID	Name	City
1	William	Vegas
2	Kevin	California
3	Nick	Texas

Employee Information (Table 2)

ID	Bonas_ID	Bonas(Dollar)
1	44	45,000
2	49	72,000
3	51	30,000

Syntax:

```
SELECT ID, Name, Bonus
FROM
table1
LEFT JOIN
table2
ON table1.ID = table2.Employee_ID
INTERSECT
SELECT ID, Name, Bonus
FROM
table1
RIGHT JOIN
table2
ON table1.ID = table2.Employee_ID;
```

EXCEPT CLAUSE

ID	Name	Bonas(Dollar)
1	William	45,000
2	Kevin	72,000
3	Nick	30,000

The inverse of the INTERSECT clause is the EXCEPT clause. In this case, all rows outside of the shared rows between the two SELECT statements are included in the result.

The syntax is as follows:

```
SELECT column-1, column-2 ……
FROM table 1
WHERE…..
EXCEPT
SELECT column-1, column-2 ……
FROM table 2
WHERE…..
```

Citing an example from the above example of Employee Information (Table 1) and Employee Information (Table 2)

Query:

```
SELECT ID, Name, Bonus
FROM
table1
LEFT JOIN
table2
ON table1.ID = table2.Employee_ID
EXCEPT
SELECT ID, Name, Bonus
FROM
table1
RIGHT JOIN
table2
ON table1.ID = table2.Employee_ID;
```

USING CLAUSE

3	Nick	Null

If numerous columns have the same names but different datatypes, the NATURAL JOIN clause can be changed with the USING clause to designate which columns should be used for an EQUIJOIN. When more than one column matches, the USING Clause is used to match only one of them.

- The terms NATURAL JOIN and USING CLAUSE are mutually exclusive.
- There should be no qualifiers (table name or Alias) in the referred columns.
- NATURAL JOIN joins tables by using all columns with matching names and datatypes. The USING Clause allows you to define which columns should be utilised for an EQUIJOIN.

Syntax:

```
SELECT <table_name>.<column_name> AS <column_name>
    FROM <table_name> JOIN <table_name> USING
(<column_name>)
```

As an illustration, consider the two tables that follow in this section.

Pet_ID	Animal	Name	Owner ID
101	Dog	Rika	123
202	Cat	Charms	134
303	Mouse	Mick	123
404	Parrot	Howdy	156
505	Rabbit	Smith	134
606	Hamster	Muffin	156

Table: Owner

Owner ID	Nick
123	Tiru
134	Tarous
156	Surfit

Query: To learn the names of the pet's owners.

```
SELECT Owner.name AS owners, PETS.name AS pet, PETS.
Animal
FROM Owner JOIN PETS USING (ownerid);
```

Output:

Owners	Pet	Animal
Tiru	Rocky	Dog
Tarous	Charles	Cat
Tiru	Mickey	Mouse
Surfit	Honey	Parrot
Tarous	Smudge	Rabbit
Surfit	Fluffy	Hamster

KNOWING HOW TO USE THE SQL MERGE STATEMENT

The SQL MERGE clause is a popular clause that allows you to manage inserts, updates, and deletes all in one transaction without having to create separate logic for each. You can specify, among other things, the circumstances in which the MERGE statement should insert, update, or delete data. You have additional possibilities for changing your intricate SQL scripts while also increasing readability when you use the MERGE SQL function. The MERGE command updates an existing table based on the results of a key field comparison with another table in the context (Figure 2.1).

The accompanying diagram demonstrates the basic operation of a SQL MERGE command. As you can see, two circles – referred to as a Source and a Target, respectively – represent two tables. The MERGE command compares the source and destination tables based on a key field and then does some processing. The MERGE statement combines the INSERT, UPDATE, and DELETE actions into a single statement. Although the MERGE statement is more complicated than basic INSERTs or UPDATEs, if you understand the fundamental principle, you'll be able to utilise it more frequently than individual INSERTs or UPDATEs.

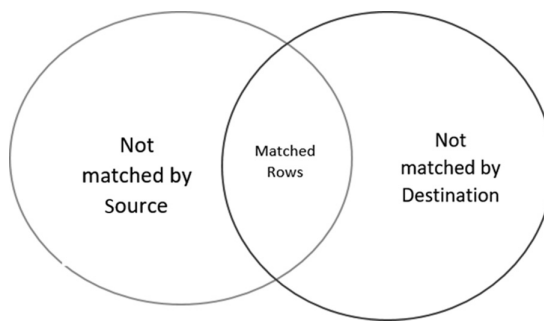


FIGURE 2.1 Illustration of merge statement.

SQL MERGE COMMAND AND ITS APPLICATIONS

Maintaining a history of data in the warehouse with a reference to the source data given to the ETL tool is frequently necessary in a standard SQL Data warehouse system. This is a typical use case when trying to handle Slowly Changing Dimensions (SCD) in the data warehouse. In such cases, updating the values of records in the warehouse whose values have changed in the source and deleting or flagging records that are no longer in the data source are required. With the addition of the SQL MERGE command in SQL Server 2008, database programmers may now more easily implement the logic necessary to conduct SCD in ETL while also streamlining their cumbersome code surrounding the INSERT, UPDATE, and DELETE commands.

IMPROVING THE SQL MERGE STATEMENT'S PERFORMANCE

By concentrating on a few criteria, you may be able to enhance the efficiency of your MERGE statements. You can now create a single statement that contains all of your DML statements as a consequence (INSERT, UPDATE, and DELETE). From the perspective of data processing, this is highly beneficial because it reduces the number of disc I/O operations for each of the three assertions and now only reads data once from the source.

The indexes that are utilised to match both the source and target tables have a significant impact on how quickly the MERGE command executes. Optimising join conditions is essential in addition to using indexes. Additionally, we ought to consider filtering the source table so that the statement only retrieves the records necessary for the actions.

MERGE STATEMENT IN SQL EXPLAINED

As mentioned in the last essay, the MERGE statement in SQL is a mix of three INSERT, DELETE, and UPDATE commands. So, if a Source table and a Target table need to be merged, the MERGE statement can do all three actions (INSERT, UPDATE, and DELETE) at the same time.

The use of the MERGE Statement will be demonstrated with a simple example.

Let's pretend there are two tables:

PRODUCT LIST is a table that includes current information on available items, with fields Prd_ ID, Prd _NAME, and Prd_ PRICE relating to each product's ID, name, and price.

UPDATED LIST is a table that holds updated information about the products, with fields Upd_ ID, Upd_ NAME, and Upd_ PRICE relating to the product's ID, name, and price.

Product List:

Prd_ID	Prd_Name	Prd_Price
10	Sauces	22.00
12	Bakery	30.00
14	Tea	45.00

Updated List:

Prd_ID	Prd_Name	Prd_Price
10	Sauces	25.00
12	Bakery	39.00
17	Juices	75.00

The goal is to update the PRODUCT LIST's product data to match the UPDATED LIST.

Solution

Let's break down this example into steps to make it easier to understand.

Step 1: Recognise the TARGET and SOURCE tables.

Because the PRODUCT LIST is the TARGET table and the UPDATED LIST is the SOURCE table in this example, the PRODUCT LIST will serve as the TARGET table and the UPDATED LIST will act as the SOURCE table.

Step 2: Recognise the tasks that must be completed.

As can be observed, there are three mismatches between the TARGET and SOURCE tables, which are as follows:

- The price of COFFEE in TARGET is 15.00, but it is 25.00 in SOURCE.

```
     PRODUCT_LIST
12     Bakery    30.00

     UPDATED_LIST
12     Bakery    39.00
```

- SOURCE does not have a tea product, but TARGET does.

```
PRODUCT_LIST
14      tea      45.00
```

- TARGET does not have a Juices product, but SOURCE does.

```
UPDATED_LIST
17      Juices 75.00
```

As a result of the foregoing inconsistencies, three operations must be performed in the TARGET. They are as follows:

1. UPDATE operation

```
10      SAUCE      25.00
```

2. Operation DELETE

```
14        TEA       45.00
```

3. the INSERT command

```
104      JUICES    75.00
```

Step 3: Compose a SQL query.

```
/* Selecting the Target and the Source */
MERGE PRODUCT_LIST AS TARGET
      USING UPDATE_LIST AS SOURCE
      /* 1. Performing the UPDATE operation */
      /* If the Prd_ID is same,
      check for change in Prd_NAME or Prd_PRICE */
      ON (TARGET. Prd_ID = SOURCE.Prd_ID)
      WHEN MATCHED
            AND TARGET. Prd_NAME <> SOURCE.Prd_NAME
            OR TARGET. Prd_PRICE <> SOURCE.Prd_PRICE
      /* Update the records in TARGET */
      THEN UPDATE
            SET TARGET. Prd_NAME = SOURCE.Prd_NAME,
            TARGET. Prd_PRICE = SOURCE.Prd_PRICE
      /* 2. executing the INSERT */
      /* Insert the records into the target table
```

```
 when there are no records that match the TARGET
table. */
      WHEN NOT MATCHED BY TARGET
      THEN INSERT (Prd_ID, Prd_NAME, Prd_PRICE)
            VALUES (SOURCE.Prd_ID, SOURCE.Prd_NAME,
SOURCE.Prd_PRICE)
      /* 3. Executing  the DELETE  */
      /* eliminate the records
from the target table when no records match the SOURCE
table. */
      WHEN NOT MATCHED BY SOURCE
      THEN DELETE
/* END OF MERGE */
```

Output: PRODUCT_LIST

Prd_ID	Prd_Name	Prd_Price
10	Sauces	25.00
12	Bakery	39.00
17	Juices	75.00

So, with the help of the MERGE statement, we can conduct all three primary statements in SQL at the same time.

DDL, DML, DCL AND TCL COMMANDS

Structured Query Language (SQL), as we all know, is a database language that allows us to execute specific operations on existing databases as well as construct new databases. SQL performs the essential activities by using commands such as Create, Drop, and Insert.

These SQL commands are divided into four types, namely, Data Modification Language (DML), Data Definition Language (DDL), Data Control Language (DCL), and Transaction Control Language (TCL). Details about individual commands are already being described in Chapter 1.

Create Domain in SQL

This statement can be used to create and define domains. CREATE DOMAIN is a command that creates a new domain. A domain is just a data type with constraints that can be applied (restrictions on the allowed

set of values). The user who creates a domain becomes the owner of that domain. The domain is formed in the supplied schema if a schema name is specified (for example, CREATE DOMAIN myschema.mydomain...).

If not, it will be created in the current schema. The domain name must be distinct from the other kinds and domains in the schema. Domains are useful for storing and maintaining common field constraints in a single location. For example, numerous tables may contain email address columns, all of which require the same CHECK constraint to ensure that the address syntax is correct. Rather of setting up each table's constraint individually, define a domain.

Parameters: There are various parameters in create domain:

✓ `Name`

The domain's name that will be created (optionally schema-qualified).

✓ `Data type`

The domain's underlying data type. Array specifiers are an example of this.

✓ `Collation`

A collation for the domain that is optional. If no collation is supplied, the default collation of the underlying data type is utilised. If COLLATE is given, the underlying type must be collatable.

✓ `EXPRESSION BY Default`

The default value is provided via the DEFAULT clause for columns with the domain data type. Any variable-free expression can be used as the value (but subqueries are not allowed). The default expression's data type must match the domain's data type. If no default value is supplied, the null value is used as the default.

Any insert operation that does not specify a value for the column will use the default expression. Any default connected with the domain is overridden if a default value is provided for a specific column. As a result, any default value associated with the underlying data type is overridden by the domain default.

✓ `NOT NULL`

This domain's values aren't allowed to be null.

✓ `NULL`

This domain's values are allowed to be null. This is the default behaviour. This clause is only for compatibility with SQL databases that aren't standard. Its use in new applications is discouraged.

✓ `CHECK (expression)`

The CHECK clauses define the integrity restrictions or tests that the domain's data must pass. Each constraint must be a Boolean result-producing expression. The key term VALUE should be used to refer to the value being examined. TRUE or UNKNOWN evaluating expressions succeed. If the result of the expression is FALSE, an error is reported, and the value cannot be changed to the domain type. CHECK expressions can't have subqueries or refer to variables other than VALUE right now. When there are several CHECK constraints in a domain, they are tested in alphabetical order by name.

Create a New Domain

In order to create a new domain, follow the steps below (SQL)

- Make a connection to a database.
- Use the CREATE DOMAIN statement to create a new domain.

Simple domains (example 1)

Some of the database's columns will be used to store people's names, while others will be used to hold addresses. The type following domains can then be defined.

```
CREATE DOMAIN persons_Name CHAR (30)
CREATE DOMAIN street_Address CHAR (35)
```

After you've specified these domains, you may use them in the same way that you'd use the built-in data types. These definitions, for example, can be used to create the following tables.

```
CREATE TABLE customer (
   id INT  DEFAULT AUTOINCREMENT  PRIMARY KEY
   name persons_name
   address street_address
)
```

Example 2: Default values, check restrictions, and identifiers.

The table's primary key is declared to be of type integer in the example above. Many of your tables may require identifiers that are identical. Creating an identifier domain for usage in these applications is significantly more convenient than saying that they are integers.

You can define a default value and a check constraint when creating a domain to ensure that no improper values are typed into any column of this kind.

Table identifiers are frequently made up of integer values. Positive numbers are a solid choice for unique IDs. Because such identifiers are likely to be used in multiple tables, the following domain could be defined:

```
CREATE DOMAIN identifier INT
DEFAULT AUTOINCREMENT
CHECK ( @col > 0 )
```

The variable @col is used in this check constraint. You can rewrite the above-mentioned definition of the customer table using this definition.

```
CREATE TABLE customer (
   id identifier PRIMARY KEY
   name persons_name
   address street_address
)
```

Example 3: Built-in domains

Some domains are predefined in Adaptive Server Anywhere. You can use these predefined domains in the same way that you would a custom domain. The following monetary domain, for example, has already been constructed for you.

Syntax:

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
where constraint is:
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

Examples:
The Indian postal code data type is created in this example, and it is then used in a table definition. To ensure that the value appears like a genuine Indian postal code, a regular expression test is used:

```
CREATE DOMAIN Indian_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);
CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal Indian_postal_code NOT NULL
);
```

DESCRIBE STATEMENT

The SQL DESCRIBE TABLE command is responsible for providing information about a certain table in the database. To display the structure of a database table or tables on the server, we'll use the SQL command DESCRIBE or another term DESC, which is equal to DESCRIBE. We'll use either DESCRIBE or DESC to retrieve information about the table in the database and locate the properties associated with it. Both are Case Insensitive and yield comparable results. We use the DESCRIBE TABLE query to acquire information on the name of the column, its data type, its NULL or NOT NULL properties, and the table's database size accuracy, as well as the If NUMERIC type scale.

Syntax:

```
DESCRIBE one;
  OR
DESC one;
```

Note that we can use DESCRIBE or DESC (both are Case Insensitive). Assume our table, named one, has three columns: F_NAME, L_NAME, and SALARY, each of which can have null values.

Output:

Name	Null	Type
F_NAME		CHAR (25)
L_NAME		CHAR (25)
SALARY		NUMBER (6)

We may see the structure of a table using DESC or DESCRIBE, but not on the console tab; the structure of a table is visible in the describe tab of the Database System Software. So, the desc or describe command displays the table's structure, which includes the column name, data type, and nullability, which indicates whether the column may store null values or not. All of these characteristics of the table are described at the time of its creation.

Consider the following scenario:

- Creating a table or specifying a table's structure

```
create table one
(
S_id int not null,
S_name char(25)
)
```

We built a table with the name one and the columns S_ID, S_NAME, and S_ID is of the non-null type, which means we can't put null values in the S_ID column but can in the S_NAME field.

To explain DESC, consider the following example:

- Defining the table's structure, or creating a table:

```
create table one
(
 S_id int not null,
 S_name char(25),
 city varchar2(25)
)
```

- Showing the table's structure

```
DESC one
  OR
DESCRIBE one
```

Output:

Name	Null	Type
S_ID	Not Null	INT
S_NAME		CHAR (25)
CITY		VARCHAR2 (25)

The ID field is not null; however, the other two columns can have null values. We must use the DESC command solely on your system software; it will not work in any editor. Make sure you're only running this command on your own database.

CASE STATEMENT IN SQL

If you need to conditionally add a value to a cell dependent on other cells, you'll use SQL's case statement.

The case statement in SQL is equivalent to an if statement or a switch statement in other languages. It enables you to define a value in a conditional manner, changing the cell's value according to whether the condition is satisfied.

Syntax of SQL Case Statements

The syntax is complex, but it is still simple: the term CASE indicates the start of a case statement, and the keyword END indicates the end. Then, you may enter the keyword WHEN followed by the condition that must be met for a single condition. The term THEN is followed by the value for that condition, such as WHEN condition> THEN stuff>.

Other WHEN/THEN statements can be added after that. With the ELSE keyword, you can add a value to use by default if none of the criteria are true, as shown below.

```
CASE
   WHEN condition1 THEN stuff
   WHEN condition2 THEN other stuff
   ...
   ELSE default stuff
END
```

Let's put this into practise to better comprehend it.

Examples of SQL Case Statements are as follows:

Let's look at an example of the CASE statement. There is a table with a list of students and their exam results. We need to assign a grade to each student, and we can do so automatically with the case statement.

ID	Name	Score
1	Simon	60
2	Iris	80
3	Maliki	52
4	Chrissy	98
5	Lemma	84
6	Alex	82
7	Uranus	69
8	Raza	78
9	Calvin	87
10	Alvin	57
11	Gabriela	89
12	Marcel	99
13	Christopher	55
14	Nick	81
15	Elvin	71
16	Leo	90
17	Johnny	90
18	Anais	90
19	Ryan	97
20	Simpson	61
21	Elena	63
22	Kathrin	51

We may use the CASE statement to assign a grade to each student, which will be stored in a new column called grade. Let's start by writing the CASE statement, which will include the grade breakdown. If the score is 94 or higher, the row will be given the value of A. If the score is 90 or greater, it will be assigned the letter A-, and so on.

```
CASE
    WHEN score >= 94 THEN "A++"
    WHEN score >= 90 THEN "A+"
    WHEN score >= 87 THEN "B++"
    WHEN score >= 83 THEN "B+"
    WHEN score >= 80 THEN "B-"
    WHEN score >= 77 THEN "C++"
    WHEN score >= 73 THEN "C+"
    WHEN score >= 70 THEN "C-"
    WHEN score >= 67 THEN "D++"
    WHEN score >= 60 THEN "D+"
    ELSE "F"
  END
```

We'll use the CASE statement in a query after we've written it. Then, using the AS keyword, we'll give the column the name grade:

```
SELECT *,
  CASE
        WHEN score >= 94 THEN "A++"
    WHEN score >= 90 THEN "A+"
    WHEN score >= 87 THEN "B++"
    WHEN score >= 83 THEN "B+"
    WHEN score >= 80 THEN "B-"
    WHEN score >= 77 THEN "C++"
    WHEN score >= 73 THEN "C+"
    WHEN score >= 70 THEN "C-"
    WHEN score >= 67 THEN "D++"
    WHEN score >= 60 THEN "D+"
    ELSE "F"
  END AS grade
FROM students_grades;
```

The table returned by this query looks like this – each student now has a grade based on their performance.

ID	Name	Score	Grade
1	Simon	60	D+
2	Iris	80	B-
3	Maliki	52	F
4	Chrissy	98	A++

(*Continued*)

ID	Name	Score	Grade
5	Lemma	84	B++
6	Alex	82	B+
7	Uranus	69	D++
8	Raza	78	B-
9	Calvin	87	B++
10	Alvin	57	F
11	Gabriela	89	B++
12	Marcel	99	A++
13	Christopher	55	F
14	Nick	81	B-
15	Elvin	71	C-
16	Leo	90	A+
17	Johnny	90	A+
18	Anais	90	A+
19	Ryan	97	A++
20	Simpson	61	D+
21	Elena	63	D+
22	Kathrin	51	F

Example 1 of a Case Statement: We can, for example, use ORDER BY to arrange the rows so that the highest grades appear first.

```
SELECT name,
  CASE
    WHEN score >= 94 THEN "A++"
    WHEN score >= 90 THEN "A+"
    WHEN score >= 87 THEN "B++"
    WHEN score >= 83 THEN "B+"
    WHEN score >= 80 THEN "B-"
    WHEN score >= 77 THEN "C++"
    WHEN score >= 73 THEN "C+"
    WHEN score >= 70 THEN "C-"
    WHEN score >= 67 THEN "D++"
    WHEN score >= 60 THEN "D+"
    ELSE "F"
  END AS grade
FROM students_grades
ORDER BY score DESC;
```

Because the alphabetical order is not the same as the order of the grades based on their worth, we rank by score, which is a number, rather than the

grade column. To render it in descending order, we use the DESC keyword, with the highest value at the top.

The table that we obtain looks like this:

Name	Grade
Simon	D+
Iris	B-
Maliki	F
Chrissy	A++
Lemma	B++
Alex	B+
Uranus	D++
Raza	B-
Calvin	B++
Alvin	F
Gabriela	B++
Marcel	A++
Christopher	F
Nick	B-
Elvin	C-
Leo	A+
Johnny	A+
Anais	A+
Ryan	A++
Simpson	D+
Elena	D+
Kathrin	F

Example 2 of a Case Statement: Let us look at these numbers in more detail. To count how many kids received each grade, we can use GROUP BY and COUNT.

```
SELECT
  CASE
     WHEN score >= 94 THEN "A++"
     WHEN score >= 90 THEN "A+"
     WHEN score >= 87 THEN "B++"
     WHEN score >= 83 THEN "B+"
     WHEN score >= 80 THEN "B-"
     WHEN score >= 77 THEN "C++"
```

```
      WHEN score >= 73 THEN "C+"
      WHEN score >= 70 THEN "C-"
      WHEN score >= 67 THEN "D++"
      WHEN score >= 60 THEN "D+"
      ELSE "F"
  END AS grade,
  COUNT(*) AS number_of_students
FROM students_grades
GROUP BY grade
ORDER BY score DESC;
```

Because score is a number, we use Sort BY to order the grades from highest to lowest (because ordering by the grade column would employ an alphabetical order, which is different from the order of the grades by value).

Grade	Number_of_Students
A++	3
A-	3
B++	2
B+	1
B-	3
C+	1
C-	1
D+	1
D	3
F	4

The case statement is a helpful tool when you need values based on specific conditions.

UNIQUE CONSTRAINTS IN SQL

To ensure that no duplicate values are put in certain columns that do not participate in a primary key, you can use SQL Server Management Studio or Transact-SQL to define a unique constraint. When you construct a unique constraint, you'll also get a matching unique index. UNIQUE constraints in SQL Server allow you to make sure that the data in a column, or a combination of columns, is unique across all rows in a table.

The following statement produces a table in which the email column has data that is unique among the entries in the Hr_people table:

```
CREATE SCHEMA Hr;
GO
```

```
CREATE TABLE Hr_persons(
    E_id INT IDENTITY PRIMARY KEY,
    F_name VARCHAR (255) NOT NULL,
    L_name VARCHAR (255) NOT NULL,
    Email VARCHAR(255) UNIQUE
);
```

The UNIQUE constraint is defined as a column constraint in this syntax. The UNIQUE constraint can also be defined as a table constraint, as shown below:

```
CREATE TABLE Hr_persons(
    E_id INT IDENTITY PRIMARY KEY,
    F_name VARCHAR (255) NOT NULL,
    L_name VARCHAR (255) NOT NULL,
    Email VARCHAR (255) UNIQUE
UNIQUE (email)
);
```

To ensure that the information stored in the columns that are a part of the UNIQUE constraint is unique, SQL Server creates a UNIQUE index in the background. As a result, attempting to insert a duplicate record will result in SQL Server rejecting the modification and returning an error message stating that the UNIQUE constraint has been violated.

The sentence below creates a new row in the Hr_persons table:

```
INSERT INTO Hr_persons (F_name, L_name, Email)
VALUES ('Joe','Johnas','J.johnas@king.burger');
```

The statement performs just as intended. However, due to the duplicate email, the following sentence fails.

```
INSERT INTO Hr_persons (F_name, L_name, Email)
VALUES ('Joe','Johnas','J.johnas@king.burger');
```

The following error message was issued by SQL Server:

```
Violation of UNIQUE KEY constraint 'UQ__persons__
LR7E617240E'. In the object "HR people," a duplicate
key cannot be inserted. The value of the second key is
(J.johnas@king.burger).
```

If you don't give the UNIQUE constraint a unique name, SQL Server will come up with one for you. The constraint name in this case is UQ people LR7E617240E, which is not very readable.

The CONSTRAINT keyword is used to assign a specific name to a UNIQUE constraint, as shown below:

```
CREATE TABLE Hr_persons (
    E_id INT IDENTITY PRIMARY KEY,
    F_name VARCHAR (255) NOT NULL,
    L_name VARCHAR (255) NOT NULL,
    Email VARCHAR (255),
    CONSTRAINT unique_email UNIQUE(email)
);
```

The benefits of giving a UNIQUE constraint a unique name include the following:

- It's a lot easier to categorise the error message now.
- When you want to change a constraint, you can refer to its name.

Contrasting the Primary Key and Unique Constraints

Despite the fact that UNIQUE and PRIMARY KEY restrictions are present data uniqueness, you should use UNIQUE instead of PRIMARY KEY when you want to enforce the uniqueness of a column or a collection of columns that aren't primary key columns.

UNIQUE constraints, unlike PRIMARY KEY constraints, allow NULL. Furthermore, because UNIQUE constraints consider NULL as a regular value, only one NULL per column is allowed.

The statement below inserts a record with a NULL value in the email column:

```
INSERT INTO hr.persons( F_name, L_name)
VALUES('Johny','Depp');
```

If you try to insert another NULL into the email column now, you'll get the following error:

```
INSERT INTO Hr_persons(F_name, L_name)
VALUES('Amber','Turd');
```

The following is the result:

```
Violation of UNIQUE KEY constraint 'UQ__persons__
LR7E617240E'. In the object "Hr people," a duplicate
key cannot be inserted. The value for the duplicate
key is (NULL>).
```

Unique Constraints for a Group of Columns

You construct a UNIQUE constraint for a group of columns as a table constraint, with column names separated by commas, as shown below:

```
CREATE TABLE table_name (
    key_column data_type PRIMARY KEY,
    column1 data_type,
    column2 data_type,
    column3 data_type,
    ...,
    UNIQUE (column1,column2)
);
```

The example below creates a UNIQUE constraint with two columns, person id and skill id:

```
CREATE TABLE hr.person_skills (
    id INT IDENTITY PRIMARY KEY,
    person_id int,
    skill_id int,
    updated_at DATETIME,
    UNIQUE (person_id, skill_id)
);
```

Add Unique Constraints to Existing Columns

When you add a UNIQUE constraint to an existing column or a group of columns in a database, SQL Server looks at the existing data in those columns to make sure that all of the values are unique. If SQL Server detects duplicate values, an error is generated and the UNIQUE constraint is not added.

The syntax for addition of a UNIQUE constraint to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
UNIQUE(column1, column2,...);
```

Assume you have the Hr_persons table as follows:

```
CREATE TABLE Hr_persons (
    E_id INT IDENTITY PRIMARY KEY,
    F_name VARCHAR (255) NOT NULL,
    L_name VARCHAR (255) NOT NULL,
    Email VARCHAR (255),
    Phone VARCHAR (20),
);
```

In the email column, the following statement introduces a UNIQUE constraint:

```
ALTER TABLE Hr_persons
ADD CONSTRAINT unique_email UNIQUE (email);
```

The following sentence, for example, adds a UNIQUE constraint on the phone column:

```
ALTER TABLE Hr_persons
ADD CONSTRAINT unique_phone UNIQUE (phone);
UNIQUE limitations should be removed.
```

The Alter Table Drop Constraint Statement
It is used to define a UNIQUE constraint like follows:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

The unique phone constraint in the Hr_persons table is removed using the following statement:

```
ALTER TABLE hr.persons
DROP CONSTRAINT unique_phone;
```

Change the Limitations that are Unique. Because there is no direct query in SQL Server to modify a UNIQUE constraint, you must drop the constraint first and then recreate it if you wish to update it.

CREATE TABLE EXTENSION

The CREATE TABLE clause in SQL has an extension that generates a new table with the same schema as an existing table in the database. It's used to temporarily store the results of difficult searches in a new table. The schema of the new table is identical to that of the referring table. The new table inherits the referring table's column names and data type by default.

Syntax:

```
CREATE TABLE newTable LIKE pets;
```

Example:

```
CREATE TABLE newTable as
            (SELECT *FROM pets
             WHERE pets.BREED = 'shitzu')
```

Queries:

Pets Table:

ID	Name	Breed	Gender
441	Tom	Lasa	Male
442	Pizu	Golden Retriever	Male
443	Simba	Poodle	Male
444	Pixel	Shitzu	Female
445	Lobby	Labrador	Female
446	Liza	German Shepherd	Male

Query 1:

```
CREATE TABLE newTable LIKE pets;
SELECT *
FROM newTable
where newTable.GENDER = 'Female';
```

Result:

ID	Name	Breed	Gender
444	Pixel	Shitzu	Female
445	Lobby	Labrador	Female

Explanation: The newTable is a duplicate of the pets table. As a result, selecting female pets from newTable only provides two rows with female pets.

Query 2:

```
CREATE TABLE newTable as
            (SELECT *
             FROM pets
             WHERE pets.BREED = 'German Shepherd');
SELECT * from newTable;
```

Output:

ID	Name	Breed	Gender
446	Liza	German shepherd	Male

Primary, the inner query is executed, then the results are saved in a new temporary relation. The outer query is then evaluated, which creates newTable and adds the output of the inner query to it.

RENAME IN SQL

Database administrators and users may want to change the name of a table in a SQL database in order to give it a more relevant name in some cases. By using the RENAME TABLE and ALTER TABLE statements in Structured Query Language, any database user can quickly modify the name. The RENAME TABLE and ALTER TABLE syntax can be used to change the table's name.

In SQL, the RENAME query has the following syntax.

```
RENAME old_table _name to new_table_name;
```

In SQL, there are several examples of the RENAME statement.

We've picked the following two SQL examples to demonstrate how to use the RENAME statement to modify the name of a SQL table in a database:

Let's look at an example of a table called Cars:

Car Name	Car Colour	Car Cost
Hyundai i20	White	1,085,000
Renault KWID	White	950,000
Toyota Fortuner	Red	900,000
Mahindra Scorpio	White	1,000,000
Kia Seltos	Black	800,000
Lamborghini Urus	Red	795,000

Table: Automobiles

- Let's say you wish to rename the above table 'Car 2022 Details'. You must type the following RENAME query in SQL to accomplish this.

```
RENAME Cars To Car_2022_Details;
```

- Following this statement, the table 'Cars' will be renamed 'Car 2021 Details'.

Let's look at an example of a table called Employee:

E_Id	E_Name	E_Salary	E_City
01	Amirah	25,000	Goa
02	Anisette	45,000	Delhi
03	Bheem	30,000	Goa
04	Rashid	29,000	Goa
05	Suyash	40,000	Delhi

Let's say you wish to use an ALTER TABLE statement to rename the aforementioned table to 'Coding Employees'. The SQL query to enter for this is as follows:

```
ALTER TABLE Employee RENAME To Coding Employees;
```

The table 'Employee' will now be known as 'Coding Employees' as a result of this statement.

ADD, DROP, MODIFY

To perform addition, removal, or modification of columns in an existing table, use the ALTER TABLE command. It is also used to add and remove constraints from an existing table.

Table Change – Add

ADD is used to insert new columns into an existing table. When we need to add more information, we don't have to recreate the entire database; instead, ADD comes to our rescue.

Syntax:

```
ALTER TABLE table_name
              ADD (Columnname_1  datatype,
              Columnname_2  datatype,
              …
              Columnname_n  datatype);
```

Change Table – Drop

The DROP COLUMN command is used to remove a column from a table. The unneeded columns are removed from the table.

Syntax:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Modify the Table

It is used to make changes to the existing columns in a table. Multiple columns can also be changed at the same time. The syntax of different databases may differ slightly.

Syntax:

```
ALTER TABLE table_name
ALTER COLUMN column_name column_type;
```

Let us understand it using an example:

Sample Table: Student

Roll_No.	Name
1	Ram
2	Abhi
3	Tanu
4	Rahul

Query:

- To ADD 2 columns AGE and COURSE to table Student.
  - `ALTER TABLE Student ADD (AGE number(3),COURSE varchar(40));`

Output:

Roll_No.	Name	Age	Course
1	Ram		
2	Abhi		
3	Rahul		
4	Tanu		

- MODIFY the COURSE column in the Student table.

```
ALTER TABLE Student MODIFY COURSE varchar(20);
```

After running the preceding query, the maximum size of the Course Column is reduced from 40 to 20.

- REMOVE the column COURSE from the Student table.

```
ALTER TABLE Student DROP COLUMN COURSE;
```

Output:

Roll_No.	Name	Age
1	Ram	
2	Abhi	
3	Rahul	
4	Tanu	

LIMIT CLAUSE

One of the most crucial clauses in SQL is LIMIT. Developers must filter their scripts when working with enormous databases in order to get to the precise number. This is the function of LIMIT, which aids in providing you with the necessary outcomes in the manner you desire, in a well filtered manner.

A Limit Clause: What Is It?

Large tables benefit greatly from the SQL LIMIT clause, which makes it simple to code for multi-page results or SQL pagination. When they are returned, numerous records can have an impact on success. Only a few tuples can be

shown at once even if the question requirements are met for many of them. The LIMIT clause limits the number of tuples that SQL can return.

- Not all SQL versions, it should be noted, support this clause.
- You can also define the SQL2008 OFFSET/FETCH FIRST clause.
- For LIMIT/offset expressions, there must be a non-negative integer.

The SQL LIMIT operator can be used when you need to pick the top three students in a class without using any conditional statements.

- When using the LIMIT x OFFSET y command, the initial y inputs are saved before the next x input is returned.
- Offset will only be used in ORDER BY clauses. You can't see it by yourself.

The value of OFFSET must be larger than zero or equal to it. Negative feedback is not an option since errors would then recur.

Give an Example to Clarify: The LIMIT clause in MySQL controls how many logs are returned.

If you decide to choose any record between 1 and 30 from the 'Orders' table, the SQL query will look like this:

```
$sql = "SELECT * FROM Orders LIMIT 30";
```

When the aforementioned SQL query was run, they returned the first 30 records.

What if you wish to choose 16 to 25 records (including)?

Using OFFSET in MySQL is another option.

The SQL statement "return only 10 records, launch record 16 (OFFSET 15)" reads as follows: `$sql = "SELECT * FROM Orders LIMIT 10 OFFSET 15";`

What are the Definition, Syntax, and Parameter Values of a Select Limit Statement?

The SELECT parameter uses the LIMIT clause to LIMIT the number of records to return. One or two claims are accepted by the LIMIT clause. Both of the statements' values must either be true or false.

The LIMIT clause syntax is demonstrated by the next two arguments:

```
SELECT

select list

FROM

table name

LIMIT [offset], "row count;"
```

Let's say we are related, Student.

Student Table:

Roll No.	Name	Grade
1200	Adi	7
1202	Suhail	8
1203	Himani	9
1204	Rob	10
1205	Simar	11
1206	Annei	12
1207	Yusufa	13
1208	Ali	14

Queries:

```
SELECT *
FROM Student
LIMIT 5;
```

Output:

1201	**Adi**	**9**
1202	Suhail	6
1203	Himani	8
1204	Rob	9
1205	Sita	7

```
SELECT *
FROM Student
ORDER BY Grade DESC
LIMIT 3;
```

Output:

12006	**Anne**	**10**
12001	Aditya	9
12004	Robin	9

When we need to identify the top three students in a class but do not want to use any conditional statements, we can utilise the LIMIT operator in cases like the one described above.

Parameters or Arguments

- **Expressions**: The computations or columns that you want to get back.
- **Tables:** The tables from which you want to get records. The FROM clause must list one table at a minimum.
- **Where Conditions**: the prerequisites that must be satisfied in order to choose the records.
- **Order by Expression**: It is used in the SELECT LIMIT statement to help you target the records you want to return and order the results. Ascending order is ASC, and descending order is DESC.
- **Limit Number_Rows**: It determines the maximum number of result set rows depending on number rows that will be returned. With LIMIT 10, for instance, the first 10 entries that satisfy the SELECT criteria would be returned. Sort order is important in this situation, therefore employ an ORDER BY clause wisely.
- **Offset_Value**: LIMIT will use offset value to select which row to return first.

Using the Limit Keyword

The LIMIT keyword restricts the number of rows in a result set that are returned.

- The LIMIT number can be any number between 0 and 255. If the LIMIT is set to zero (0), no rows are returned from the set result.

- You can choose which line to start the data recovery on using the OFFSET value.
- The Syntax of LIMIT works well with the Choose Update or DELETE order.

When Should the Limit Clause Be Used?

Consider that you are developing a programme that makes use of the MyFlixDB database. The system designer instructed you to limit the amount of records displayed on a screen to 20 records in order to combat slow load times. How should the framework that satisfies those customer needs be implemented? The LIMIT keyword is helpful in these circumstances. The data returned from the inquiry can be limited to just 20 documents per tab.

The Limit Clause's Benefits

Only the maximum amount of rows are included in the result collection, according to the LIMIT clause (or exactly the maximum rows, in the event that max row count is less than the number of qualifying rows). There are no additional rows returned that satisfy the question collection condition. It can also set the maximum value using a host variable or a local variable set to the SPL input value. The rows returned are sorted by ORDER BY if the LIMIT clause comes after the ORDER BY clause. ORDER BY may only be effective for searches that return a subset of the qualified rows by limiting the order of rows because query results are typically not delivered in a precise order.

INSERT IGNORE STATEMENT

The INSERT statement is used to carry out this action when inserting data into the tables of the database. The IGNORE keyword can be added to the INSERT statement syntax. It is usually recommended to use the INSERT IGNORE statement rather than the INSERT statement. This is so that records in MySQL databases don't become inconsistent or redundant by handling errors that arise during the addition of duplicate records with the INSERT IGNORE statement.

How Does MySQL's Insert Ignore Function Work?

When using the INSERT statement to attempt to insert many records into a specific table of the MySQL database and an error occurs for some reason, MySQL will stop the execution of the query and report the error without adding any rows to the table that we attempted to insert. However,

Mysql will issue a warning and insert all the proper records while leaving out and excluding the rows that were the cause of the mistake if we use INSERT IGNORE instead of simply a simple insert command.

Syntax:
The INSERT IGNORE statement's syntax is as follows:

```
INSERT IGNORE INTO table(list_of_columns)
VALUES(record1),
(record2),
```

Where list of columns is the comma-separated names of the column that you intend to include in the record, and record1, record2, are the values of the columns you have indicated in the list of columns in the same order as they have been given in the list.

Drawback

Since certain errors could go unreported, most users do not favour INSERT IGNORE over INSERT. This might lead to discrepancies in the table, which would prevent some tuples from being inserted without giving the user an opportunity to fix them. Thus, INSERT IGNORE must only be used in very particular circumstances.

Example: Say we have a relation, Employee.

Employee Table:

Emp_ID	Name	City
1501	Aakash	Dehradun
1503	Salil	Bangalore
1509	Joe	Hyderabad
1508	Shelli	Delhi
1502	Ana	Mumbai
1504	Semma	Pune

As we can notice, the entries are not sorted on the basis of their primary key, i.e., Emp_ID.

Sample Query:

```
INSERT IGNORE INTO Employee (EmployeeID, Name, City)
VALUES (15010, 'Rameshwar', 'Mumbai');
```

Output:
No entry inserted.

Inserting Multiple Records:
When inserting multiple records at once, any that cannot be inserting will not be, but any that can will be:

INSERT IGNORE INTO Employee (EmployeeID, Name, City)
VALUES (15007, 'Shikha', 'Delhi'), (15002, 'Ram', 'Mumbai'), (15009, 'Sam', 'Ahmedabad');

Output:
The first and last entries are added; the intermediate entry is merely disregarded. No error message flashes.

As Mysql attempts to change the values to arrange them in the correct manner and inserts the correct records except the one that can create an issue, using the INSERT IGNORE statement rather than just inserting statements is always a recommended practise.

LIKE OPERATOR

To determine whether a particular character string matches a given pattern, the logical operator SQL Like is employed. It is typically utilised in a Where clause to search for a specific pattern in a column. When pattern matching is required rather than equal or not equal, this operator can be helpful. When we wish to return a row if a given character string matches a predetermined pattern, we utilise the SQL Like function. Regular characters and wildcard characters may both be used in the pattern. Regular characters must exactly match the characters supplied in the character string in order to return a row. Any portion of the character string can be matched using the wildcard characters.

Syntax:

```
SELECT * FROM table_name
WHERE column_name LIKE 'pattern'
```

The following wildcard characters can be used with the LIKE operator to specify a pattern:

Pattern	Description
%	The percentage matches 0–1, multiple capital or small-capital characters, or numbers. E.g., 'B%' will match all string starting with 'B' further followed by any number of characters or numbers.
_	Any single letter or number can be represented by the underscore (_) symbol. E.g., 'B_' will match all strings with two chars where the first character must be 'B' and second character can be anything.
[]	The [] searches any single character within the specified range in the []. E.g., 'B[e,l,p]' will match 'Ball', 'Bat', 'Bird' etc.
[^]	The [^] seaches any single character except the specified range in the [^]. E.g., 'B [^e,l,p]' will match anything that starts with 'B', but not 'Bpple', 'Belp', 'Blep', 'Bple', etc.

When using the LIKE operator, these wildcard characters can be used singly or in combination.

For example: Let us use Employee info table in all the examples:

EmpId	FirstName	LastName	Email	Salary	HireDate
1	'John'	'King'	'john.king@abc.com'	33,000	2018-07-25
2	'James'	'Bond'			2018-07-29
3	'Neena'	'Kochhar'	'neena@test.com'	17,000	2018-08-22
4	'Lex'	'De Haan'	'lex@test.com'	15,000	2018-09-8
5	'Amit'	'Patel'		18,000	2019-01-25
6	'Abdul'	'Kalam'	'abdul@test.com'	25,000	2020-07-14

Query 1:

```
SELECT *
FROM Employee
WHERE FirstName LIKE 'john';
```

The query WHERE FirstName LIKE 'john' above searches all the records in the MS SQL Server, SQLite, and MySQL databases where the value of the FirstName column is 'john' or 'John'. But with Oracle and PostgreSQL databases, the LIKE operator is case-sensitive, so it only retrieves entries where the value is 'john,' not 'John. The outcome in the MS SQL Server, SQLite, and MySQL databases is as follows.

EmpId	FirstName	LastName	Email	Salary	HireDate
1	'John'	'King'	'john.king@abc.com'	33,000	2018-07-25

Any number of characters may be specified with the wildcard char %.

```
SELECT *
FROM Employees
WHERE FirstName LIKE 'j%';
```

The aforementioned query retrieves all records from the MS SQL Server, SQLite, and MySQL databases where the value of the First Name column begins with either "j" or "J" followed by any number of characters. It will retrieve records in Oracle or PostgreSQL that begin with "j," but not "J." The outcome in the MS SQL Server, SQLite, and MySQL databases is as follows:

EmpId	FirstName	LastName	Email	Salary	HireDate
1	'John'	'King'	'john.king@abc.com'	33,000	2018-07-25
2	'James'	'Bond'			2018-07-29

The next query returns information where the value of FirstName is "% a %". It implies that "a" must appear somewhere in the value.

```
SELECT *
FROM Employee
WHERE FirstName LIKE '%a%';
```

The results of the aforementioned query are shown below:

EmpId	FirstName	LastName	Email	Salary	HireDate
2	'James'	'Bond'			2018-07-29
3	'Neena'	'Kochhar'	'neena@test.com'	17,000	2018-08-22
5	'Amit'	'Patel'		18,000	2019-01-25
6	'Abdul'	'Kalam'	'abdul@test.com'	25,000	2020-07-14

Records with a FirstName value of three letters and 'e' in the second position will be returned by the query below. Any one character is indicated by the '_'.

```
SELECT *
FROM Employee
WHERE FirstName LIKE '_e_';
```

EmpId	FirstName	LastName	Email	Salary	HireDate
4	'Lex'	'De Haan'	'lex@test.com'	15,000	2018-09-8

The [] wildcard pattern is used in the following query

```
SELECT *
FROM Employee
WHERE FirstName LIKE 'B [i,m,t,y,s]';
```

The [^] wildcard pattern is used in the following query.

```
SELECT *
FROM Employee
WHERE FirstName LIKE 'B [^i,m,t,y,s]';
```

EmpId	FirstName	LastName	Email	Salary	HireDate
6	'Abdul'	'Kalam'	'abdul@test.com'	25,000	2020-07-14

LIKE NOT

Filter records that do not match the provided string by combining the NOT operator with the LIKE operator.

```
SELECT *
FROM Employee
WHERE FirstName NOT LIKE 'j%';
```

FirstName NOT LIKE 'j%' in the above sentence retrieves entries where the FirstName values do not begin with 'j'.

EmpId	FirstName	LastName	Email	Salary	HireDate
3	'Neena'	'Kochhar'	'neena@test.com'	17,000	2018-08-22
4	'Lex'	'De Haan'	'lex@test.com'	15,000	2018-09-8
5	'Amit'	'Patel'		18,000	2019-01-25
6	'Abdul'	'Kalam'	'abdul@test.com'	25,000	2020-07-14

SOME SQL OPERATOR

When a value is compared to each value in a list of query results, SQL Server's SOME operator is used to determine whether at least one row is present in the result of an inner query. The comparison operators must

come before SOME in order for it to match at least one entry in the subquery. Consider that when SOME is used, greater than (>) signifies greater than at least one value.

Syntax:

```
SELECT [col_name... | express1 ]
FROM [t_name]
WHERE express2 comparison_operator {ALL | ANY | SOME}
( subquery )
```

Parameters:

Name	Description
Col_name	Name of the column of the table.
Express1	Expressions can be parts of a SQL query that do computations or value comparisons against other values and are composed of a single constant, variable, scalar function, or column name.
Table name	The table's name.
Where expression2	Until the SOME operator finds a match, compares a scalar expression. For the SOME operator to return a Boolean TRUE value, one or more rows must match the expression.
Comparison operator	Compares the subquery and the expression. A standard comparison operator (=, >,!=, >, >=,, or =) must be used in the comparison.

Let us understand some operators using an example:

Instructor Table:

Name	Department	Salary
Chandra	Computational Biology	1
Visweswaran	Electronics	1.5
Abraham	Computer Science	1.3
John	Electronics	1.2
Samantha	Computer Science	2
Jyoti	Electronics	1.2
Debarka	Computer Science	2
Ganesh	Computational Biology	0.9

Sample Queries and Outputs:

```
select name
from instructor
where Salary > some(select Salary
from instructor
where dept='Computer Science');
```

Output:
Visweswaran
Samantha
Debarka

Explanation:
The teachers whose salaries exceed one or more instructors' salaries in the "Computer Science" department are asked to leave. The "Computer Science" department pays 1.3, 2, and 2 in wages. This implies that any instructor earning more than 1.3 can be counted in the outcome.

OFFSET FETCH IN SQL SERVER

The usage of OFFSET FETCH capabilities in fetching a record with restricted memory and avoiding an out-of-memory exception. In SQL Server, the SELECT and ORDER BY clauses are combined with the FETCH and OFFSET clauses to specify a range of records to be returned by the query. For performing result set pagination, it was first introduced with SQL Server version 2012. When our database has a tonne of data, it is helpful.

Application of Offset and Fetch Offset

OFFSET: The starting point for returning rows from a result set is specified using this clause. In essence, it disregards the first batch of records. Only the ORDER BY clause can be used with it in SQL Server. An error will be returned if its value is negative. It must therefore always be higher than or equal to zero.

FETCH: In a query, this optional clause specifies how many rows we want to return after the OFFSET. Without OFFSET, we cannot use it. Similar to OFFSET, its value cannot be negative. As a result, it must always be bigger than or equal to zero in order to avoid an error.

Syntax:
The syntax used to demonstrate how to utilise OFFSET and FETCH clause is as follows:

```
SELECT * FROM table_name
ORDER BY columns [ASC |DESC]
OFFSET no_of_rows_to_skip
FETCH {FIRST | NEXT} no_of_rows_to_return {ROW | ROWS}
ONLY
```

We have specified the table name from which the data will be obtained in this syntax. For records shown in ascending or descending order, we then specified the ORDER BY clause. The amount of records was then skipped using the OFFSET specification, and the set of records was then returned using FETCH.

Let's look at how the OFFSET and FETCH clauses are used practically. Let's say we have "my table" with the information below:

a	b
1	100
2	99
3	98
4	97
5	96
6	95
7	94

Fetch and Offset

```
SELECT * from "my table" order by a offset 3 rows
fetch next 2 rows only;
```

Result:

a	b
4	97
5	96

Offset Only

```
Query: SELECT * from "my table" order by a offset 3
rows;
```

Results:

a	b
4	97
5	96
6	95
7	94

Fetch Only

Query:

```
SELECT * from "my table" order by a fetch next 2 rows
only;
```

Results:

a	b
1	100
2	99

SQL STATEMENT EXCEPT

When two SELECT queries are being used to select records, the SQL EXCEPT statement is one of the most often utilised statements to filter records. It returns the entries from the left SELECT query that are not present in the results given by the SELECT query on the right side of the EXCEPT statement. The way a SQL EXCEPT statement operates is quite similar to how a math minus operator operates. Microsoft SQL Server 2005 introduces the EXCEPT statement.

Prerequisites for the SQL Except Statement

Before using the EXCEPT statement in SQL Server, a few prerequisites must be satisfied:

- The tables used to execute the SELECT operations should have the same number of columns and order.

- The related SELECT queries should use data types that are compatible or the same for the corresponding columns in both tables.

Consider the following scenario: Students and TA (Teaching Assistant). All of the students who aren't teaching assistants should return, as shown in the following form of the question:

Students Table:

StudentID	Name	Course
1	Rohan	DBMS
2	Kevin	OS
3	Mansi	DBMS
4	Mansi	ADA
5	Rekha	ADA
6	Megha	OS

TA Table:

StudentID	Name	Course
1	Kevin	TOC
2	Sita	IP
3	Manik	AP
4	Rekha	SNS

Syntax:

```
SELECT Name
       FROM Students
EXCEPT
SELECT NAME
       FROM TA;
```

Output:

Rohan
Mansi
Megha

We must specifically use EXCEPTALL rather than EXCEPT in order to keep duplication.

```
SELECT Name
       FROM Students
EXCEPTALL
SELECT Name
       FROM TA;
```

Output:

Rohan
Mansi
Mansi
Megha

USING JOINS AND THE OVER CLAUSE IN SQL TO COMBINE AGGREGATE AND NON-AGGREGATE VALUES

Aggregate functions add together several numbers and give back a single result. Consider the following structure for the employee table EMP and the department table DEPT:

Table – Employee Table

Name	Name	Course
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7, 2)
COMM		NUMBER(7, 2)
DEPTNO		NUMBER(2)

Table – Department Table

Name	Null	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

Moreover, the following outcomes are required:

- Display the name, sal, and job of the emp, as well as the maximum, minimum, average, and total sal of all the employees doing the same job.
- Display the department name and the number of employees in it.

A result cannot be obtained by directly combining aggregated and non-aggregated values. Consequently, one can use the following ideas:

UTILISING JOINS

- Make a subtable that contains the values that have been aggregated.
- Use Join to display the results from the subtable with their raw values.

Solutions for problem 1 using JOIN:

```
SELECT ENAME, SAL, EMP.JOB,
            SUBTABLE.MAXSAL, SUBTABLE.MINSAL,
            SUBTABLE.AVGSAL, SUBTABLE.SUMSAL
FROM EMP
INNER JOIN
        (SELECT JOB, MAX(SAL) MAXSAL, MIN(SAL)
                MINSAL, AVG(SAL) AVGSAL, SUM(SAL)
SUMSAL
         FROM EMP
          GROUP BY JOB) SUBTABLE
                  ON EMP.JOB = SUBTABLE.JOB;
```

Output for Sample Data:

Ename	Sal	Job	MaxSal	MinSal	AvgSal	SumSal
SCOTT	3,300	ANALYST	3,300	1,925	2,841.67	8,525
HENRY	1,925	ANALYST	3,300	1,925	2,841.67	8,525
FORD	3,300	ANALYST	3,300	1,925	2,841.67	8,525
SMITH	3,300	CLERK	3,300	1,045	1,746.25	6,985
MILLER	1,430	CLERK	3,300	1,045	1,746.25	6,985

OVER CLAUSE

- Partition by and over clause are used to break down data into partitions.
- For each partition, the specified function is active.

Solutions for problem 2 using OVER Clause:

```
SELECT DISTINCT(DNAME),
COUNT(ENAME) OVER (PARTITION BY EMP.DEPTNO) EMP
FROM EMP
RIGHT OUTER JOIN DEPT
ON EMP.DEPTNO=DEPT.DEPTNO
ORDER BY EMP DESC;
```

Dname	Emp
SALES	6
RESEARCH	5
ACCOUNTING	3
OPERATIONS	0
OTHERS	0

OPERATORS FOR SQL ANY AND ALL

You can compare a single column value to a variety of other values using the ANY and ALL operators.

ANY Operator in SQL

THE ANY OPERATING SYSTEM: As a result, returns TRUE if ANY of the subquery values satisfy the condition ANY signifies that the condition will be true if the operation is true for any of the values in the range.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

SQL Operator ALL

A logical operator called the SQL ALL compares a single result to a set of values from a subquery that are returned in a single column. The SQL ALL operator's syntax is demonstrated by the following:

```
WHERE column_name comparison_operator ANY (subquery);
```

A comparison operator like >, >=, =, >, or = must come before the SQL ALL operator, and a subquery must come after it. A list of literal values may be used in place of a subquery in some database systems, such as Oracle. Keep in mind that the WHERE clause's condition is always true if the subquery returns no rows. The following table demonstrates the significance of the SQL ALL operator, assuming that the subquery produces one or more rows:

Demonstration Database

EXISTS IN SQL

One of the key SQL operators, exists, enables you to specify a subquery to check whether a specific object exists in the database. It executes the query using the syntax listed below.

Syntax:

```
SELECT [column_name... | expression1 ]
FROM [table_name]
WHERE EXISTS (subquery)
```

If the subquery returns any rows, the operator returns TRUE; otherwise, it returns FALSE. The SELECT, UPDATE, INSERT, or DELETE commands can all be used with the exists operator. The following is an explanation of the syntax's parameters.

Parameter	Description
Col_name	Name of the column
Express 1	This could be any expression composed of a single variable, constant, or even the name of a column.
t_name	The title of the column on which we are working
Where exists	It looks for the presence of one or more rows in the subquery. If there is a row, the Boolean value is TRUE; otherwise, it is FALSE

Query:

```
SELECT col_name FROM t_name
WHERE EXISTS
(SELECT col_name FROM t_name WHERE condition);
```

Value returned

It gives back a True or False Boolean value.

Example:

The Employee table in the code snippet below includes the following columns: Employee id, first name, last name, salary, and department.

```
-- Creating a worker table
CREATE table Employee(
    Employee_id int,
    first_name varchar(100),
    last_name varchar(100),
    salary int,
    department varchar(100),

PRIMARY KEY(Employee_id));

-- Creating a bonus table
CREATE table bonus(
    Employee_id int,
    bonus_amt int,

FOREIGN KEY(Employee_id) REFERENCES
Employee(Employee_id));

-- Inserting record in Employee table
INSERT INTO Employee

values (1, "Charlie", "Arora", 100000, "Engineering"),
       (2, "Niharika", "Verma", 80000, "Admin"),
       (3, "Thomas", "Martin", 300000, "HR"),
       (4, "William", "O'Brien", 500000, "Admin"),
       (5, "Vivek", "Bhati", 500000, "Admin"),
       (6, "Vipul", "Diwan", 200000, "Account"),
       (7, "Satish", "Kumar", 75000, "Account"),
       (8, "Anderson", "Lee", 90000, "Admin");
```

```
-- Inserting record in bonus table
INSERT INTO bonus

values (1, 5000),
       (2, 5500),
       (3, 4000),
       (1, 4500),
       (2, 3500);

-- This nested query will return unique employee ids
-- where employee ids in bonus table equals to
employee
-- ids in employee table and bonus amount less than
5500
SELECT Employee_id FROM Employee

WHERE EXISTS (SELECT Employee_id FROM bonus WHERE
              bonus.Employee_id = Employee.Employee_id
              AND bonus_amt < 5500);
```

Output:
Employee_id
1
2
3

GROUP BY STATEMENT IN SQL

To group similar data into groups, the SELECT statement and the SQL GROUP BY clause are employed. In a SELECT statement, this GROUP BY clause comes after the WHERE clause and before the ORDER BY clause.

Syntax:
The following code block illustrates the fundamental grammar of a GROUP BY clause. If there is an ORDER BY clause, it must come after the GROUP BY clause and come before the constraints in the WHERE clause.

```
SELECT col1, col2
FROM t_name
WHERE [ conditions]
GROUP BY col1, col2
ORDER BY col1, col2
```

Example:
Consider the following records in the CUSTOMERS table:

ID	Name	Age	Address	Salary
111	Muffin	24	Indore	10,000.00
124	Kamal	22	MP	4,500.00
126	Hardin	27	Bhopal	8,500.00
145	Chantal	25	Mumbai	6,500.00
149	Ramesh	32	Ahmedabad	2,000.00
152	Khilan	25	Delhi	1,500.00
178	kaushik	23	Kota	2,000.00

The following GROUP BY query would be used if you wanted to discover the entire amount of each customer's salary.

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

Result:

Name	Salary
Muffin	10,000.00
Kamal	4,500.00
Hardin	8,500.00
Chantal	6,500.00
Ramesh	2,000.00
Khilan	1,500.00
kaushik	2,000.00

Let's take a look at a table where the CUSTOMERS table has the records with the following names twice:

ID	Name	Age	Address	Salary
111	Muffin	24	Indore	10,000.00
124	Muffin	22	MP	4,500.00
126	Hardin	27	Bhopal	8,500.00
145	Hardin	25	Mumbai	6,500.00
149	Ramesh	32	Ahmedabad	2,000.00
152	Khilan	25	Delhi	1,500.00
178	kaushik	23	Kota	2,000.00

Once more, the GROUP BY query might be as follows if you wanted to know the total amount of salary for each customer:

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

This would produce the following result:

ID	Name	Age	Address	Salary
111	Muffin	24	Indore	10,000.00
126	Hardin	27	Bhopal	8,500.00
149	Ramesh	32	Ahmedabad	2,000.00
152	Khilan	25	Delhi	1,500.00
178	kaushik	23	Kota	2,000.00

UNION CLAUSE

As its name suggests, this operator/clause is used to join the results of two or more SELECT queries. We can also use this command to access a specific record from a certain column of the table. The number of columns in each SELECT statement and the order in which they are used in the UNION statement must match. In addition, the data type of each column must be the same in all SELECT statements. The UNION clause returns only distinct values. For good measure you need to copy the values, then you need to use the UNION ALL clause.

Syntax:

```
SELECT column_1, column_2,...column_n
FROM table_1
UNION
SELECT column_1, column_2,...column_n
FROM table_2;
```

Duplicate rows in the result table are removed using the UNION clause. To keep copies, check UNION ALL. Quite a few SELECT statements can be merged using the UNION statement, both UNION and UNION ALL can be used to join different tables. If you want to consolidate from different SELECT statements are one of a kind, on the off chance that uniqueness is not a concern, then use UNION ALL for better performance.

The following limitations apply to unions:

- The SELECT statements must return the same number of columns.
- Even though the column names need not match, the columns that the return of SELECT operations must match in order and data type.
- Individual ORDER BY clauses are not allowed in SELECT statements.

After the last SELECT statement, add an ORDER BY clause to the resulting table to sort the data. A union names the columns it returns as results based on the initial SELECT statement. Connections are often parsed from left to right by default. You can use parentheses to define an evaluation sequence. The UNION clause allows you to combine any number of SELECT statements.

Union Versus Union All

One of the major difference between UNION and UNION ALL is that UNION eliminates duplicate rows and UNION ALL does not remove duplicate rows.

In SQL, An Example of the Union Operator

Let's make two separate tables and insert records into each.

The query below generates the **Old Employee table**, which has four fields:

```
CREATE TABLE Old_Employee
(
Employee_Id INT NOT NULL,
Employee_Name Varchar (40),
Emp_Age INT,
Emp_Salary INT
);
```

The query generates the **New Employee** table, which has four fields:

CREATE TABLE New_Employee

```
(
Emp_Id INT NOT NULL,
Emp_Name Varchar (40),
```

```
Emp_Age INT,
Emp_Salary INT
);
```

The query generates the new Employee table, which has four fields:

```
INSERT INTO Old_Employee (Emp_Id, Emp_Name, Emp_Age,
Emp_Salary) VALUES (11, Akhima, 12 , 15000),
(12, Abhi, 17, 16000),
(13, Surya, 16, 19000),
(14, Abhimak, 17, 26000),
(15, Ritika, 26, 29000),
(16, Yashi, 29, 28000);
```

The following INSERT query adds a new employee record into the Old Employee table:

```
Query: SELECT * FROM Old_Employee;
```

Employee_Id	Employee_Name	Emp_Age	Emp_Salary
11	Akhima	28	25,000
12	Abhi	27	26,000
13	Surya	26	29,000
14	Abhimak	27	26,000
15	Ritika	26	29,000
16	Yashi	29	28,000

Table: Old Employee

The following INSERT query adds a new employee record to the New Emp table:

```
INSERT INTO New_Employee (Emp_Id, Emp_Name, Emp_Age,
Emp_Salary) VALUES (01, Jack, 18, 45000),
(202, Berry, 19, 35000),
(15, Ritika, 16, 29000),
(203, Shyam, 17, 26000),
(204, Ritik, 18, 38000),
(16, Yashi, 19, 28000);
```

The following query displays the New Employee table's details:

SELECT * FROM New_Emp;

Emp_Id	Emp_Name	Emp_Salary	Emp_City
201	Jack	28	45,000
202	Berry	29	35,000
15	Ritika	26	29,000
203	Shyam	27	26,000
204	Ritik	28	38,000
16	Yashi	29	28,000

Table: New_Emp

Using the UNION operator, the following query displays all data from both tables in a single table:

```
SELECT * FROM Old_Emp UNION SELECT * FROM
New_Employee;
```

Output:

Emp_Id	Emp_Name	Emp_Age	Emp_Salary
11	Akhima	28	25,000
12	Abhi	27	26,000
13	Surya	26	29,000
14	Abhimak	27	26,000
15	Ritika	26	29,000
16	Yashi	29	28,000
201	Jack	28	45,000
202	Berry	29	35,000
203	Shyam	27	26,000
204	Ritik	28	38,000

Where Clause Is Combined with the Union Operator

To filter records from one or both tables, use the WHERE clause in conjunction with the UNION operator.

UNION syntax with WHERE clause

```
SELECT Col_Name_1, Col_Name_2 ...., Col_NameN FROM
T_Name_1 [WHERE condition]
```

```
UNION
SELECT Col_Name1, Col_Name_2 ...., Col_Name_N FROM
T_Name_2 [WHERE condition];
```

UNION with WHERE Clause Example

The following query returns the records of employees from the preceding tables whose salaries are greater than or equal to $29,000:

```
SELECT * FROM Old_Emp WHERE Emp_Salary >= 29000UNION
SELECT * FROM New_Emp WHERE Emp_Salary >= 29000;
```

Output:

Emp_Id	Emp_Name	Emp_Age	Emp_Salary
13	Surya	26	29,000
15	Ritika	26	29,000
201	Jack	28	45,000
202	Berry	29	35,000
204	Ritik	28	38,000

Union All Operator in SQL

The SQL Union is same as the Union ALL Operator, yet the main distinction is that UNION ALL administrator likewise shows the normal lines in the outcome. Administrator, yet the main distinction is that UNION ALL administrator likewise shows the normal lines in the outcome.

Syntax of Union All Set operator:

```
SELECT Col_Name_1, Col_Name_2 ...., Col_Name_N FROM
T_Name_1 [WHERE condition]
UNION ALL
SELECT Col_Name_1, Col_Name_2 ...., Column_Name_N FROM
Table_Name_2 [WHERE condition];
```

Example of Union All

Let's make two separate tables and insert records into each. The query below generates the Passed Students database, which has four fields:

```
CREATE TABLE Passed_Students
(
Stud_Id INT NOT NULL,
Stud_Name Varchar (40),
Stud_Age INT,
Stud_Marks INT
);
```

The query below generates the New Students table, which has four fields:

```
CREATE TABLE New_Students
(
Stud_Id INT NOT NULL,
Stud_Name Varchar (40),
Stud_Age INT,
Stud_Marks INT
);
```

The INSERT query below adds a record of passed students to the Passed Students table:

```
INSERT INTO Passed_Students (Stud_Id, Stud_Name, Stud_
Age, Stud_Marks) VALUES (11, Akhima, 28, 95),
(12, Abhi, 27, 86),
(13, Surya, 26, 79),
(14, Abhimak, 27, 66),
(15, Ritika, 26, 79),
(16, Yashi, 29, 88);
```

The following query displays the Passed Students table's details:

```
SELECT * FROM Passed_Students;
```

Stud_Id	Stud_Name	Stud_Age	Stud_Marks
11	Akhima	28	95
12	Abhi	27	86
13	Surya	26	79
14	Abhimak	27	66
15	Ritika	26	79
16	Yashi	29	88

Table: Passed Students

The following INSERT query adds a new student record to the New Students table:

```
INSERT INTO New_Students (Stud_Id, Stud_Name, Stud_
Age, Stud_Marks)  VALUES (201, Jack, 28, 77),
(202, Berry, 29, 68),
(15, Ritika, 26, 82),
(203, Shyam, 27, 70),
(204, Ritik, 28, 99),
(16, Yashi, 29, 86);
```

The following query displays the New Students table's details:

```
SELECT * FROM New_Students;
```

Stud_Id	Stud_Name	Stud_Age	Stud_Marks
201	Jack	28	77
202	Berry	29	66
15	Ritika	26	82
203	Shyam	27	70
204	Ritika	28	99
16	Yashi	29	86

Table: New Students

The query below returns all duplicate and unique records from both tables:

```
SELECT * FROM Passed_Stud UNION ALL SELECT * FROM
New_Stud;
```

Output:

Stud_Id	Stud_Name	Stud_Age	Stud_Marks
11	Akhima	28	95
12	Abhi	27	86
13	Surya	26	79
14	Abhimak	27	66
15	Ritika	26	79

(*Continued*)

Stud_Id	Stud_Name	Stud_Age	Stud_Marks
16	Yashi	29	88
201	Jack	28	77
202	Berry	29	68
15	Ritika	26	82
203	Shyam	27	70
204	Ritik	28	99
16	Yashi	29	86

SQL IN ALIASES

Aliases are temporary names assigned to tables or columns for the sake of a certain SQL query. It is used when the name of a column or table is changed from its original name, but the change is just temporary.

- Aliases are used to make table or column names easier to read.
- The renaming is only a temporary alteration, and the table name in the original database remains unchanged.
- When table or column names are long or difficult to understand, aliases come in handy.
- These are preferred when a query involves more than one table.

Syntax:
As a Column Alias:

```
SELECT column as alias_name FROM table_name;
column: fields in the table
alias_name: the temporary alias name to be used in
replacement of original column name
table_name: name of table
```

Example of Aliasing a Column Name

Aliases are typically used to make the column heads in your result set easier to read. Most frequently, you will alias a column in your query when using an aggregate function such as MIN, MAX, AVG, SUM, or COUNT.

We should examine an illustration of how to pseudonym a segment name in SQL.

In this model, we have a table called representatives that contains the accompanying data:

Emp_Number	l_Name	f_Name	Salary	Dept_ID
101	Smithy	Johnny	62,000	50
102	Ander	Janae	57,500	50
103	Ever	Brady	71,000	51
104	Horvath	Jacky	42,000	51

Let's go over how to alias a column. Enter the SQL statement below:

```
SELECT dept_id, COUNT(*) AS total
FROM employees
GROUP BY dept_id;
```

There will be two records chosen. You should see the following outcomes:

Dept_ID	Total
50	2
51	2

We've aliased the COUNT (*) field as total in this example. As a result, when the result set is returned, total will appear as the heading for the second column. Because our alias name does not contain any spaces, we do not need to wrap it in quotes.

Change our query such that the column alias has a space:

```
SELECT dept_id, COUNT(*) AS "total employees"
FROM employees
GROUP BY dept_id;
```

There will be two records chosen. You should see the following outcomes:

Dept_ID	Total Employees
50	2
51	2

In this example, we've aliased the COUNT(*) field to 'total workers', which will serve as the header for the second column in our result set. Because this column alias contains spaces, 'total employees' must be contained in quotes in the SQL statement.

As a Table Alias:

```
SELECT column FROM table_name as alias_name;
column: fields in the table
table_name: name of table
alias_name: temporary alias name to be used in
replacement of original table name
```

A Table Name Aliasing Example

When you alias a table, you either want to shorten the table name to make the SQL statement more concise and understandable, or you want to utilise the same table name in the FROM clause more than once (i.e., self-join).

Let's look at how a table name can be aliased in SQL.

In this illustration, we have a table called products with the following data:

Product_ID	Product_Name	Category_ID
1	Pear	50
2	Banana	50
3	Orange	50
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

And, a table called categories, which has the following information:

Category_ID	Category_Name
25	Deli
50	Produce
75	Bakery
100	General Merchandise
125	Technology

Let us now combine these two tables and alias the table names. Enter the SQL statement as follows:

```
SELECT p.product_name, c.category_name
FROM products AS p
INNER JOIN categories AS c
ON p.category_id = c.category_id
WHERE p.product_name <> 'Pear';
```

There will be five records chosen. You should see the following outcomes:

Product_Name	Category_Name
Banana	Produce
Orange	Produce
Apple	Produce
Bread	Bakery
Sliced Ham	Deli

We've built an alias for the goods table and an alias for the categories table in this example. We may now refer to the products table as p and the categories table as c within this SQL expression.

It is not essential to generate aliases for all of the tables provided in the FROM clause when establishing table aliases. You have the option of creating aliases on any or all of the tables.

ORDER BY CLAUSE IN SQL

Depending on one or more columns, the ORDER BY clause in SQL is used to sort data in either ascending or descending order. By default, some databases sort query results in ascending order.

Syntax:
The basic syntax is as follows:

```
SELECT col-list
FROM t_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

The ORDER BY clause might include more than one column. Make sure the column you're sorting by is in the column list.

Example:
Consider the CUSTOMERS table, which contains the following records:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
7	Muffin	24	Indore	10,000.00

An example is provided in the following code block, which sorts the results in ascending order by NAME and SALARY.

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME, SALARY;
```

This would yield the following result:

ID	Name	Age	Address	Salary
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
3	kaushik	23	Kota	2,000.00
2	Khilan	25	Delhi	1,500.00
6	Kamal	22	MP	4,500.00
7	Muffin	24	Indore	10,000.00
1	Ramesh	32	Ahmedabad	2,000.00

An example is provided in the following code block, which sorts the result in descending order by NAME.

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

This would yield the following result:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
7	Muffin	24	Indore	10,000.00
6	Kamal	22	MP	4,500.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
5	Hardin	27	Bhopal	8,500.00
4	Chantal	25	Mumbai	6,500.00

SELECT TOP CLAUSE IN SQL

We may want to extract a certain quantity of records from a SQL table rather than all table records. The WHERE provision can be utilised to confine the number of lines returned by a SELECT explanation. Assume you have a customer table and wish to obtain records for customers from a specific country. There could be a large number of records that meet the criteria. To meet the conditions, we need top 'N' customer records. TOP or ROW Number () clauses can be used to meet this need.

Top Clause Syntax in SQL Server

```
SELECT TOP Expression | Percentage [Column_Names] [
WITH TIES ]
FROM [Table_Name]
```

Expression: We can use a numeric expression to define the number of rows to be returned.

Percent: In this case, we can use the PERCENT value to return the percentage number of rows from the total number of rows in the output.

With Ties: The WITH TIES clause can be used to return more rows with values that match the previous row in the result set. For example, we want the top ten consumers in terms of purchase value. If we do not utilise WITH TIES, SQL Server provides exactly 10 records, despite the fact that we may have additional customers with comparable purchase costs. In this situation, we can retrieve more rows by using WITH TIES. We must use WITH TIES in conjunction with the ORDER BY Clause.

By supplying the number of records to return as the result set, we will attempt to comprehend the syntax for utilising SQL SELECT TOP.

Syntax:
SELECT TOP number column name FROM table name WHERE condition; the syntax above retrieves data from all columns depending on the WHERE clause and is limited by the number provided as part of the SELECT TOP.

Consider the following Customer Table to learn how to use the SELECT TOP command to copy all of the columns' data depending on a criterion.

Cust_Id	Cust_Name	Cust_Age	Cust_Gender
1	John	31	M
2	Amit	25	M
3	Annie	35	F
4	Tom	38	M

Scenario: Retrieve the first row of data from the Customer table where the gender is male.

```
SELECT TOP 1 * FROM Customer WHERE CustomerGender = 'M';
```

Output:

Cust_Id	Cust_Name	Cust_Age	Cust_Gender
1	John	31	M

SQL Select the Highest Percentage of Records to Return

By specifying the percentage of records to return as the result set, we will attempt to comprehend the syntax for utilising SQL SELECT TOP.

Syntax:

```
SELECT TOP number PERCENT column_name FROM table_name
WHERE condition;
```

The syntax above retrieves data from all columns based on the WHERE clause and is limited by the percent provided as part of the SELECT TOP.

Consider the following Customer Table to learn how to use the SELECT INTO command to copy all of the columns' data depending on a criterion.

Scenario: Get data from the Customer table for 50% of the entries where the gender is male.

Query:

```
SELECT TOP 50 PERCENT * FROM Customer WHERE Cust_
Gender = 'M';
```

Cust_Id	Cust_Name	Cust_Age	Cust_Gender
1	John	31	M
2	Amit	25	M

Multiple Select Top Statements

To achieve our goal, we can combine numerous SELECT TOP commands.

Syntax:

```
SELECT TOP number column_name FROM table_name WHERE
condition (the select statement with another SELECT
TOP);
```

Scenario:
Retrieve the first row of data from the Customer table where the gender is male.

Query:

```
SELECT TOP 1 * FROM Customer WHERE Cust_Age = (SELECT
TOP 1 Cust_Age FROM Customer ORDER BY Cust_Age desc);
```

Output:

Cust_Id	Cust_Name	Cust_Age	Cust_Gender
4	Tom	38	M

SQL UPDATE COMMAND

Databases are used to store data in tables. After writing data to such tables, it is usual to need to update individual fields at some point throughout the data's existence. To accomplish this, we can use the SQL UPDATE command. An Update query is a type of action query that performs numerous updates on many records at the same time.

In SQL, the UPDATE order is utilised to alter or change existing records in a table If we wish to update a specific value, we use the WHERE clause in conjunction with the UPDATE clause. On the off chance that the WHERE proviso isn't utilised, all columns will be impacted. Furthermore, depending on our requirements, we can utilise the UPDATE statement to update a single or several columns.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN =
valueN
WHERE [condition];
```

where the keywords are UPDATE, SET, and WHERE table name is the name of the table that has to be updated, col1, col2,... are the columns val1, val2,... considered to be updated? assign new values, and the condition section, which is followed by a semicolon, contains the condition.

Example:
Consider the CUSTOMERS table, which contains the following records:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
7	Muffin	24	Indore	10,000.00

The query below will change the ADDRESS for a customer whose ID number in the table is 6.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

The CUSTOMERS table would now contain the following records:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	Pune	4,500.00
7	Muffin	24	Indore	10,000.00

If you wish to change all of the ADDRESS and SALARY column values in the CUSTOMERS table, you may use the UPDATE query instead, as illustrated in the following code block.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

The CUSTOMERS table would now contain the following records:

ID	Name	Age	Address	Salary
1	Ramesh	32	Pune	1,000.00
2	Khilan	25	Pune	1,000.00
3	kaushik	23	Pune	1,000.00
4	Chantal	25	Pune	1,000.00
5	Hardin	27	Pune	1,000.00
6	Kamal	22	Pune	1,000.00
7	Muffin	24	Pune	1,000.00

DELETE STATEMENT IN SQL

To erase lines from a table, utilise the SQL DELETE explanation. The DELETE statement often deletes one or more records from a table.

Syntax:

```
DELETE FROM table_name WHERE some_condition;
table_name: name of the table
some_condition: condition to choose particular record.
```

Example:
Take a look at the 'Customer' table.

The number of records we can delete will depend on the criteria we give in the WHERE clause. If we leave out the WHERE clause, all of the records are destroyed and the table is empty.

Table: Customer

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
7	Muffin	24	Indore	10,000.00

- Single record deletion: Remove all rows where NAME='Ram.' This will only erase the first row.

Query:

```
DELETE FROM Customer WHERE NAME = 'Muffy ';
```

Result: The above query deletes only the first row, and the table Customer now looks like this:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00

- **Multiple Record Deletion**: Delete the rows in the Customer table where the Salary is 2000.

Query:

```
DELETE FROM Customer WHERE Salary = 2000.00;
```

Results: The query above will eliminate two rows (first and third), and the table Customer will now look like this:

ID	Name	Age	Address	Salary
2	Khilan	25	Delhi	1,500.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00

- **Delete Every Single Record**: There are two queries that can be used to accomplish this, as illustrated below.
  Query 1: "DELETE FROM Student";
  Query 2: "DELETE * FROM Student";
  Output: The table's records will all be deleted; no records will be displayed. Customer's table will soon be empty.

INSERT INTO SQL STATEMENT

To insert a new row into a table, use the SQL INSERT INTO statement. For inserting rows, the INSERT INTO statement can be used in two ways:

- **Simply Values**: The first technique is to specify only the data value to be added without specifying the column names.

Syntax:

```
INSERT INTO t_name VALUES (value1, value2, value3,…);
t_name: name of the table.
value1, value2,.. : value of first col, second col,…
for the new record
```

- **Both Column Names and Values**: In the second approach, we will specify both the columns to be filled and their associated values, as shown below:

Syntax:

```
INSERT INTO table_name (col1, column2, column3,..)
VALUES ( value1, value2, value3,..);
table_name: name of the table.
column1: name of first column, second column …
value1, value2, value3 : value of first column, second
column,… for the new record
```

Table Customer:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00

Table Lateral Customer:

ID	Name	Age	Address	Salary
7	Ravi	35	Kolkata	2,900.00
8	Ashish	45	Dharamshala	3,500.00
9	Kamal	43	Jaipur	2,080.00

Query 1: Method 1 (all rows and columns inserted):

```
  INSERT INTO Customer SELECT * FROM Lateral
Customer;
```

Output: The data from the table LateralCustomer will be inserted into the table Customer using this query. Student's table will now look like this:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
7	Ravi	35	Kolkata	2,900.00
8	Ashish	45	Dharamshala	3,500.00
9	Kamal	43	Jaipur	2,000.00

- **Method 2 (Particular Column Inserting)**:

```
INSERT INTO Student (ID,NAME,Age) SELECT ID,
NAME,Age FROM LateralCustomer;
```

Output: This query will insert data into the columns ROLL NO, NAME, and Age of the table Lateral Student in the table Student, with the remaining columns in the Student table filled with null, which is the default value for the remaining fields. Student's table will now look like this:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
7	Ravi	35	null	null
8	Ashish	45	null	null
9	Kamal	43	null	null

- Choose which rows to insert:

```
INSERT INTO Customer SELECT * FROM Lateral
Customer WHERE Salary = 2000;
```

Output: This query will only insert the last row from table Lateral customer into table customer. Customer's table will now look like this:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Kota	2,000.00
4	Chantal	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
9	Kamal	43	Jaipur	2,000.00

- Using a single SQL statement to insert several entries into a table

```
INSERT INTO table _ name(Col1,Col2,Col3,.......)
VALUES (Value1, Value2,Value3,.....),
         (Value1, Value2,Value3,.....),
          (Value1, Value2,Value3,.....),
          ..........................;
```

t_name: name of the table

Col1: name of first column, second column …

Value1, Value2, Value3: Value of first column, second column,… for each new row inserted.

Multiple lists of values, each separated by ',' must be provided. Every value list corresponds to the values that should be added to each new table row.

The following list of values instructs data to be entered in the table's next row.

AND AND OR SQL OPERATORS

The AND and OR operators are used in SQL to filter data and produce exact results based on requirements. Multiple conditions can also be combined using the SQL AND & OR operators. These two operators can be combined in a SELECT, INSERT, UPDATE, or DELETE query to test for various conditions. It is critical to use parenthesis when combining these conditions so that the database knows what order to examine each condition in:

- The WHERE clause employs the AND and OR operators.
- Conjunctive operators are the names given to these two operators.

AND Operator

This operator returns only records where both condition1 and condition2 evaluate to True.

Syntax:

```
SELECT * FROM table_name WHERE condition1 AND
condition2 and ...conditionN;

table_name: name of the table
condition1,2,..N: first condition, second condition
and so on
```

OR Operator

This operator returns records where either of the conditions condition1 or condition2 is True. That is, either condition 1 or condition 2 is true.

Syntax:

```
SELECT * FROM table_name WHERE condition1 OR
condition2 OR... conditionN;

table_name: name of the table
condition1,2,..N : first condition, second condition
and so on
```

Now we'll look at a table database to demonstrate AND and OR operators with various cases:

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2,000.00
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Delhi	2,000.00
4	kaushik	25	Mumbai	6,500.00
5	Hardin	27	Bhopal	8,500.00
6	Kamal	22	MP	4,500.00
7	Ravi	35	Kolkata	2,900.00
8	Ashish	45	Dharamshala	3,500.00
9	Kamal	43	Delhi	2,000.00

Assume we wish to retrieve all records from the Student table where the Salary is ≤3,000 and the ADDRESS is Delhi. The question will then be as follows:

Query:

```
SELECT * FROM Student WHERE Salary ≤ 3000 AND ADDRESS
= 'Delhi';
```

Output:

ID	Name	Age	Address	Salary
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Delhi	2,000.00
9	Kamal	43	Delhi	2,000.00

As another model, assume you need to get every one of the records from the Student table where the NAME is Kamal and the Salary is 2,000.

Query:

```
SELECT * FROM Student WHERE Salary = 2000 AND NAME =
'Kamal';
```

Output:

ID	Name	Age	Address	Salary
9	Kamal	43	Delhi	2,000.00

To recover all records from the Student table where the NAME is Kamal or kaushik.

Query:

```
SELECT * FROM Student WHERE NAME = 'Kamal' OR NAME =
'Kaushik';
```

Output:

ID	Name	Age	Address	Salary
3	kaushik	23	Delhi	2,000.00
4	kaushik	25	Mumbai	6,500.00
9	Kamal	43	Delhi	2,000.00

Combining AND and OR

To build complex queries, we can combine AND and OR operators as shown below.

Syntax:

```
SELECT * FROM table_name WHERE condition1 AND
(condition2 OR condition3);
```

Consider retrieving all data from the Student table where the Age is 18 and the NAME is Ram or RAMESH.

Query:

```
SELECT * FROM Student WHERE Salary ≥ 5000 AND (Address
= 'Delhi' OR Address = 'MP');
```

Output:

ID	Name	Age	Address	Salary
2	Khilan	25	Delhi	1,500.00
3	kaushik	23	Delhi	2,000.00
6	Komal	22	MP	4,500.00
9	Kamal	43	Delhi	2,000.00

CLAUSE WHERE

The SQL WHERE statement is utilised to determine a condition while recovering information from a solitary table or interfacing numerous tables. Only if the provided condition is met does it return a specific value from the table. You should utilise the WHERE clause to filter the records and retrieve only those that are required. The WHERE clause is utilised not just in the SELECT statement, but also in the UPDATE and DELETE statements.

Syntax:
The following is the basic syntax of the SELECT statement with the WHERE clause:

```
SELECT column1, column2, column N
FROM table_name
WHERE [condition]
```

A condition can be specified using comparison or logical operators such as >, =, LIKE, and NOT. The accompanying models can assist you with figuring out this idea.

Example:
Consider the CUSTOMERS table, which contains the following records:

ID	Name	Age	Address	Salary
1	Ravi	32	Ahmedabad	2,000.00
2	Kaushik	25	Delhi	1,500.00
3	Kevin	23	Delhi	2,000.00
4	Kula	25	Mumbai	6,500.00
5	Hendrik	27	Bhopal	8,500.00
6	Kiwi	22	MP	4,500.00

The code beneath is an illustration of how to recover the ID, Name, and Salary information from the CUSTOMERS table where the compensation is bigger than 5000.

Query:

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 5000;
```

Output:

ID	Name	Salary
4	Kula	6,500.00
5	Hendrik	8,500.00
6	Kiwi	4,500.00

An example query would retrieve the ID, Name, and Salary fields from the CUSTOMERS table for a client named kiwi.

It is vital to notice that all strings should be enclosed in single quotes (''). Numeric values, on the other hand, should be presented without any quotation marks, as seen in the preceding example.

SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME='Kiwi';

The output will be as follows:

ID	Name	Salary
6	Kiwi	4,500.00

UNIQUE CLAUSE IN SQL

To delete duplicate columns from the result set, use the SQL DISTINCT clause. The unique keyword is used in conjunction with the choose keyword. It is useful in avoiding duplicate values in specific columns/tables. When we utilise the distinct keyword, we get the unique values.

- Only distinct (different) values are returned by SELECT DISTINCT.
- The DISTINCT function removes duplicate records from the table.
- DISTINCT can be used with aggregates such as COUNT, AVG, MAX, and so on.
- DISTINCT only works on a single column.
- DISTINCT does not allow multiple columns.

Syntax:

```
SELECT DISTINCT column1, column2
FROM table_name
```

column 1 and column 2 are the names of the table's fields.

table name: The name of the table from which we wish to retrieve the records.

This query will return all of the distinct combinations of rows in the table with the fields column1, column2.

If the distinct keyword is used with multiple columns, the distinct combination will be shown in the result set.

Example:

Consider the CUSTOMERS table, which contains the following records:

ID	Name	Age	Address	Salary
1	Ravi	32	Ahmedabad	2,000.00
2	Kaushik	25	Delhi	1,500.00
3	Kevin	23	Delhi	2,000.00
4	Kaushik	25	Delhi	1,500.00
5	Hendrik	27	Bhopal	8,500.00
6	Kaushik	25	Delhi	1,500.00

Queries:

- Obtaining unique names from the NAME field –

```
SELECT DISTINCT NAME
FROM Student;
```

Name
Ravi
Kaushik
Kevin
Kula
Hendrik
Kiwi

- To obtain a one-of-a-kind combination of rows from the entire table,

```
SELECT DISTINCT *
FROM Student;
```

Output:

ID	Name	Age	Address	Salary
1	Ravi	32	Ahmedabad	2,000.00
2	Kaushik	25	Delhi	1,500.00
3	Kevin	23	Delhi	2,000.00
5	Hendrik	27	Bhopal	8,500.00

Without the term distinct, six entries would have been fetched instead of four, because the original table contains six records with duplicate values.

SELECT IN SQL STATEMENT

The most widely used command in Structured Query Language is the SELECT statement. It's used to get data from one or more database tables and views. It also obtains the selected data that meets the criteria we specify.

Where Clause in Select Statement

It is used in combination with the SELECT statement to retrieve only those entries from the table that satisfy the query's given condition.

Example: A SELECT statement with a WHERE clause.

First, we must build a new table and then populate it with dummy information. To create the Employee Details table in SQL, run the following query:

SELECT Statement Syntax with WHERE Clause

```
CREATE TABLE Emp_Details
(
Empl_ID INT AUTO_INCREMENT PRIMARY KEY,
Emp_Name VARCHAR (50),
Emp_City VARCHAR (20),
Emp_Salary INT NOT NULL,
Emp_Panelty INT NOT NULL
) ;
```

The following INSERT query inserts an employee entry into the Employee Details table:

```
INSERT INTO Empl_Details (Empl_ID, Emp_Name, Emp_City,
Emp_Salary, Emp_Panelty) VALUES (11, Anupam,
Ghaziabad, 25000, 500),
```

```
(12, Tus, Lucknow, 29000, 1000),
(13, Vive, Kolkata, 35000, 500),
(14, Shiva, Goa, 22000, 500);
```

The data from the Employee Details table is shown in the following SELECT query:

```
SELECT * FROM Emp_Details;
```

Emp_Id	Emp_Name	Emp_City	Emp_Salary	Emp_Panelty
11	Anupam	Ghaziabad	25,000	500
12	Tus	Lucknow	29,000	1,000
13	Vive	Kolkata	35,000	500
14	Shiva	Goa	22,000	500

The following query returns the list of employees from the preceding table whose Emp Panelty is 500:

```
SELECT * FROM Emp_Details WHERE Emp_Panelty = 500;
```

This SELECT query returns the following table:

Emp_Id	Emp_Name	Emp_City	Emp_Salary	Emp_Panelty
11	Anupam	Ghaziabad	25,000	500
13	Vive	Kolkata	35,000	500
14	Shiva	Goa	22,000	500

Group by Clause in SQL Select Statement

The GROUP BY condition is utilised related to the SELECT explanation to show the section's normal information:

SELECT Statement Syntax with GROUP BY Clause is as follows:

```
SELECT col_Name_1, col_Name_2....., col_Name_N
aggregate_function_name(col_Name2) FROM table_name
GROUP BY col_Name1;
```

Select Statement with Group by Clause Example

To create the Cars Details table, run the following query:

```
CREATE TABLE Cars_Details
(
Car_Number INT PRIMARY KEY,
Car_Name VARCHAR (50),
Car_Price INT NOT NULL,
Car_AmountINT NOT NULL
) ;
```

The following INSERT query adds a car entry to the Cars Details table:

```
INSERT INTO Cars_Details (Car_Number, Car_Name, Car_
Amount, Car_Price)
VALUES (2578, Creta, 3, 1500000),
(9258, Audi, 2, 3000000),
(8233, Venue, 6, 900000),
(6214, Nexon, 7, 1000000);
```

The values in the output are displayed by the following SELECT query:

```
SELECT * FROM Cars_Details;
```

Car_Number	Car_Name	Car_Amount	Car_Price
2578	Creta	3	1,000,000
9258	Audi	2	900,000
8233	Venue	6	900,000
6214	Nexon	7	1,000,000

The following SELECT query with GROUP BY returns the number of autos with the same price:

```
SELECT COUNT (Car_Name), Car_Price FROM Cars_Details
GROUP BY Car_Price;
```

The outcome of the preceding GROUP BY query is displayed below.

Count (Car_Name)	Car_Price
2	1,000,000
2	900,000

Having Clause in SQL Select Statement

The SELECT statement's HAVING clause creates a selection inside the groups defined by the GROUP BY clause.

Syntax of SELECT Statement with HAVING clause is as follows:

```
SELECT col_Name_1, col_Name_2, ....., col_Name_N
aggregate_function_name(column_Name_2) FROM table_name
GROUP BY column_Name1 HAVING ;
```

SELECT Statement with HAVING Clause Example

Let's use the CREATE command below to create the Employee Having table in SQL:

```
CREATE TABLE Emp_Having
(
Emp_Id INT PRIMARY KEY,
Emp_Name VARCHAR (50),
Emp_Salary INT NOT NULL,
Emp_City VARCHAR (50)
) ;
```

The following INSERT query installs an employee record into the Employee Having table:

```
INSERT INTO Emp_Having (Emp_Id, Emp_Name, Emp_Salary,
Emp_City)
VALUES (01, Jonas, 20000, Goa),
(02, Basanti, 40000, Delhi),
(03, Rasheta, 80000,Jaipur),
(04, Aunpam, 20000, Goa),
(05, Sumitra, 50000, Delhi);
```

The following SELECT query shows the values of Emp_Having table in the output:

```
SELECT * FROM Emp_Having;
```

Emp_Id	Emp_Name	Emp_Salary	Emp_City
01	Jonas	20,000	Goa
02	Basanti	40,000	Delhi
03	Rasheta	80,000	Jaipur
04	Anupam	20,000	Goa
05	Sumitra	50,000	Delhi

The following query displays the total salary of employees with more than 5,000 in the Employee Having table:

```
SELECT SUM (Emp_Salary), Emp_City FROM Emp_Having
GROUP BY Emp_City HAVING SUM(Emp_Salary)>5000;
```

The following table is displayed by this HAVING query with a SELECT statement:

Output:

Sum (Emp_Salary)	Emp_City
90,000	Delhi
80,000	Jaipur

Order by Clause in Select Statement

With the SQL SELECT statement, the ORDER BY clause sorts the records or rows. The ORDER BY phrase orders the values ascending and descending. By default, few database systems arrange column values in ascending order.

SELECT Statement Syntax with ORDER BY Clause is as follows:

```
SELECT Col_Name_1, Col_Name_2, ....., col_Name_N FROM
t_name WHERE [Condition] ORDER BY[col_Name_1, col_
Name_2, ....., column_Name_N asc | desc ];
```

Example of SELECT Statement with ORDER BY clause in SQL

```
CREATE TABLE Employee_Order
(
Id INT NOT NULL,
FirstName VARCHAR (50),
Salary INT,
City VARCHAR (50)
) ;
```

The following INSERT query installs an employee record into the Employee Having table:

```
INSERT INTO Emp_Order (Id, FirstName, Salary, City)
VALUES (01, Jonas, 20000, Goa),
(02, Basanti, 15000, Delhi),
(03, Rasheta, 80000,Jaipur),
(04, Aunpam, 90000, Goa),
(05, Sumitra, 50000, Delhi);
```

The following SELECT query displays the table values in the output:

```
SELECT * FROM Emp_Order;
```

Id	First_Name	Salary	City
01	Jonas	20,000	Goa
02	Basanti	15,000	Delhi
03	Rasheta	80,000	Jaipur
04	Anupam	90,000	Goa
05	Sumitra	50,000	Delhi

The following query sorts the salaries of employees from the above Employee Order table in descending order:

```
SELECT * FROM Employee_Order ORDER BY Emp_Salary DESC;
```

This SQL query returns the following table:

Output:

Emp_Id	Emp_Name	Emp_Salary	Emp_City
04	Anupam	90,000	Goa
03	Rasheta	80,000	Jaipur
05	Sumitra	50,000	Delhi
01	Jonas	20,000	Goa
02	Basanti	15,000	Delhi

DROP AND TRUNCATE TABLE IN SQL

The DROP order in SQL is utilised to erase the whole data set or table files, and information, and that's only the tip of the iceberg. The TRUNCATE order, then again, is utilised to eliminate all columns from a table.

Truncate Table in SQL

Begin by issuing the TRUNCATE TABLE command. When you wish to empty a SQL table, you'll use TRUNCATE TABLE. In other words, when you wish to erase all of the data from a database but keep the table itself. Why would you want to preserve a table that doesn't have any data? Because you won't have to CREATE the structure again: you won't have to supply column names or data types.

The SQL TRUNCATE command comes in helpful in data science, for example, when you discover a flaw in your automatic data loading scripts that populate your SQL tables. A common example of such an issue is when a daily data load occurs twice. The simplest solution is to empty the table and reload the right data.

Another popular use case is when working with million-line tables with fields that must be updated daily. Instead of changing each row individually, it is sometimes more efficient to empty the table and then reload the modified data.

By the way, the syntax is really simple. It goes

```
TRUNCATE TABLE table_name;
```

TRUNCATE TABLE is the SQL command, and table name is the name of the existing table that you want to empty.

Example:
Consider the following records in a CUSTOMERS table:

ID	Name	Age	Address	Salary
1	Ravi	32	Ahmedabad	2,000.00
2	Kaushik	25	Delhi	1,500.00
3	Kevin	23	Delhi	2,000.00
4	Kula	25	Mumbai	6,500.00
5	Hendrik	27	Bhopal	8,500.00
6	Kiwi	22	MP	4,500.00

Here's an example of a Truncate command.

```
SQL >TRUNCATE TABLE CUSTOMERS;
```

The CUSTOMERS table has now been shortened, and the output of the SELECT operation is as seen in the code block below.

```
SQL>SELECT * FROM CUSTOMERS
```

Set is empty (0.00 sec)

Table Drop in SQL

DROP TABLE is our second command. Unlike TRUNCATE TABLE, DROP TABLE deletes the SQL table rather than just emptying it. It implies that you will destroy its structure, previously defined column names, data types, and, of course, the data contained within it.

Regardless, it is sometimes required. In data science, we will use it when you update your entire database structure and no longer need some of your SQL tables. Or when you inadvertently construct a table you don't want to use. (It happens, people.) Also, when you mistype something in a table (table name, column name, data type), it's sometimes easier and faster to DROP it than to fix it with ALTER TABLE.

The syntax for removing a SQL table is as follows:

```
DROP TABLE table_name;
```

Let's look at the syntax for removing the table from the database.

```
DROP TABLE table_name;
```

We would first verify the STUDENTS table before deleting it from the database.

```
SQL> DESC STUDENTS;
```

Field	Type	Null	Key	Default	Extra
ID	Int(11)	NO	PRI		
NAME	Varchar(20)	NO			
AGE	Int(11)	NO			
ADDRESS	Varchar(25)	YES		NULL	

This indicates that the STUDENTS table is present in the database, therefore we may drop it as follows:

```
SQL>DROP TABLE STUDENTS;
```

Presently, utilise the accompanying order to check regardless of whether the table exists.

```
SQL> DESC STUDENTS;
```

Output: Query OK, 0 rows affected (0.01 sec)

As you can see, the table has been dropped, therefore it is not displayed.

DROP TABLE is the command, and table name is the name of the table to be deleted. Here's a nice warning now that you know how to empty or delete a table in SQL. Be extremely cautious when using the SQL TRUNCATE TABLE or DROP TABLE commands! Yes, if you use SQL Workbench (or pgadmin4), you must commit your changes before they become irreversible... Aside than that, there is no undo button. Deleting or emptying a table can result in significant – and unpleasant – data loss, so think twice before running them. Particularly in real-world projects. Particularly if you don't have a backup of your data.

CREATE IN SQL

In SQL, there are two CREATE statements:

- CREATE TABLE
- CREATE DATABASE

Make a Database

A database is a systematic collection of data. To store data in a well-structured manner, the first step with SQL is to establish a database. In SQL, the CREATE DATABASE statement is used to create a new database.

Syntax:

```
CREATE DATABASE database_name;
database_name: name of the database.
```

Example Query: This query will establish a new SQL database and call it my database.

```
CREATE DATABASE my_database;
```

Table Creation

We learned about database creation before. To save the data, we will need a table. In SQL, the Construct TABLE statement is used to create a table. We all know that a table is made up of rows and columns. So, while constructing tables, we must give SQL with information such as the names of the columns, the type of data to be stored in the columns, the quantity of the data, and so on. Let us now go through how to utilise the Construct TABLE statement to create tables in SQL.

Syntax:

```
CREATE TABLE table_name
(
column1 data_type(size),
column2 data_type(size),
column3 data_type(size),
....
);
```

Table_Name: name of the table.

column1 name of the first column.

Data_Type: Type of data we want to store in the particular column.

For example, int for integer data.

Size: Size of the data we can store in a particular column. For example, if for a column, we specify the data_type as int and size as 15, then this column can store an integer number of maximum 15 digits.

This query will generate a table called Students with three columns: ROLL NO, NAME, and SUBJECT.

```
CREATE TABLE Students
(
ROLL_NO int(3),
NAME varchar(20),
SUBJECT varchar(20),
);
```

This query will generate the table Students. The ROLL NO field is of type int and can hold a three-digit integer value. The following two columns, NAME and SUBJECT, are of the type varchar and may store characters, with the size 20 indicating that these two fields can hold a maximum of 20 characters.

JOINS IN SQL

A JOIN clause is used to join rows from two or more tables based on a common column.

In this article, we will go over the last two JOINS:

- **Cartesian Join:** The CARTESIAN JOIN is also known as the CROSS JOIN. Each row of one table is joined to every row of another table in a CARTESIAN JOIN. This typically occurs when no matching column or WHERE criteria is supplied.
  - The CARTESIAN JOIN and CARTESIAN PRODUCT both behave similarly in the absence of a WHERE condition. In other words, the sum of the rows in the two tables makes up the number of rows in the result-set.
  - In general, a cross join is comparable to an inner join in that the join-condition always evaluates to True.

Syntax:

```
SELECT table1.column1 , table1.column2, table2.
column1...
FROM table1
CROSS JOIN table2;
table1: First table.
table2: Second table
```

- **Self-Join:** A table is joined to itself with SELF JOIN, as the name implies. That is, based on the circumstances, each row of the table is connected to itself and all other rows. In other words, it's a join between two copies of the same table.

Syntax:

```
SELECT a.coulmn1 , b.column2
FROM table_name a, table_name b
WHERE some_condition;

table_name: Name of the table.
some_condition: Condition for selecting the rows.
```

ALTERNATE QUOTE OPERATOR

Character literals are used in many SQL statements as expressions or conditions. If the literal contains a single quotation mark, you can use the quote (q) operator and specify your own delimiter. You can use any appropriate delimiter, single-byte or multibyte, or any of the character pairings [], (), {}, or >.

OPERATOR FOR CONCATENATION

The generic SQL expression diagram illustrates the syntax of the binary concatenation operator. Use the concatenation operator (||) to combine two expressions that evaluate to character or numeric data types.

Let us have a look at an example:

Syntax:

```
SELECT id, f_name, l_name, f_name || last_name,
               salary, first_name || salary FROM
myTable
```

Output (Third and Fifth Columns show values concatenated by operator ||)

id salary

id	f_name	l_name	f_name||last_name	salary	first_name||salary
101	Rajat	Rawat	RajatRawat	10,000	Rajat10000
120	Geeks	ForGeeks	GeeksForGeeks	20,000	Geeks20000
156	Shane	Watson	ShaneWatson	50,000	Shane50000
145	Kedar	Jadhav	KedarJadhav	90,000	Kedar90000

In the example above, we used the || Concatenation operator, which is used to link two or as many columns as you like in a select query and is independent of column datatype. We've linked two columns above: first name+last name and first name+salary.

Literals can also be used in the concatenation operator. Let's have a look:

Exemplification 1: Using Character Literal

Syntax:

```
SELECT id, first_name, last_name, salary,
      first_name||' has salary '||salary as "new" FROM
myTable
```

Output: (Concatenating three values and giving a name 'new')

id	first_name	last_name	salary	new
101	Rajat	Rawat	10,000	Rajat has salary10,000
120	Geeks	ForGeeks	20,000	Geeks has salary 20,000
156	Shane	Watson	50,000	Shane has salary 50,000
145	Kedar	Jadhav	90,000	Kedar has salary 90,000

In our choice statement above, we utilised has salary as a character literal. Similarly, we can use a number literal or a date literal depending on our needs.

OPERATOR MINUS

In SQL, the Minus Operator is used with two SELECT statements. The MINUS operator is used to deduct the first SELECT query's result set from the second SELECT query's result set. In other words, the MINUS operator will return only those rows that are unique in only the first SELECT query and not those that are common to both the first and second SELECT searches.

Basic Syntax:

```
SELECT column1 , column2 , ... columnN
FROM table_name
WHERE condition
MINUS
SELECT column1 , column2 , ... columnN
FROM table_name
WHERE condition;
columnN: column1, column2.. are the name of columns of
the table.
```

Important Point:

- In the given query, the WHERE clause is optional.
- Both SELECT statements must have the same amount of columns.
- Both SELECT statements' associated columns must have the same data type.

DIVISION OPERATOR

In SQL, the division operator is an arithmetic operator. Addition (+), subtraction (–), multiplication (*), division (/), and modulus are the arithmetic operators (%).

In SQL, the division operator has the following syntax:

```
SELECT <expression> / <expression>
FROM table
[WHERE expression]
```

The division operator is illustrated in the following example:

Query:

```
SELECT 4 / 2
```

You may run this query, and it will return the result, which in this example is 2. However, you are more likely to be working with numbers stored in columns in your database tables.

THE NOT OPERATOR IN SQL

It can be used before any conditional statement to select rows that meet the condition.

Syntax:
The NOT condition in SQL has the following syntax:

```
NOT condition
```

Using NOT with the IN Condition as an example

Let's begin by looking at how NOT can be used with the IN condition. We establish a NOT IN condition when we apply the NOT operator with the IN condition. This will check to see if an expression isn't already in a list. In this example, we have a table called products that contains the following information:

Product_ID	Product_Name	Category_ID
1	Pear	50
2	Banana	50
3	Orange	50

(*Continued*)

Product_ID	Product_Name	Category_ID
4	Apple	50
5	Bread	75
6	Sliced Ham	25
7	Kleenex	NULL

Enter the SQL statement below:

```
SELECT *
FROM products
WHERE product_name NOT IN ('Pear', 'Banana', 'Bread');
```

There will be four records chosen. You should see the following outcomes:

Product_ID	Product_Name	Category_ID
3	Orange	50
4	Apple	50
6	Sliced Ham	25
7	Kleenex	NULL

This example returns all entries from the products table where the product name does not match Pear, Banana, or Bread. It is sometimes more economical to mention the values that you do not want rather than the ones that you do want. It corresponds to the following SQL statement:

```
SELECT *
FROM products
WHERE product_name <> 'Pear'
AND product_name <> 'Banana'
AND product_name <> 'Bread';
```

Output:

Product_ID	Product_Name	Category_ID
3	Orange	50
4	Apple	50
6	Sliced Ham	25
7	Kleenex	NULL

BETWEEN AND IN OPERATOR

SQL provides many techniques for getting meaningful information from many types of data. Sometimes, we need to extract a range of values from all the column values in a table. For example, to retrieve information about all employees of a company who were born in a specific year, we need to use various operators and clauses to retrieve that data from databases. Here is one such operator that is used to perform such task.

Between Operator

The SQL Between operator is used to determine whether an expression falls inside a certain value range. This operator is inclusive, meaning it includes the range's start and end values. The values can be textual, numeric, or date based. This operator is compatible with the SELECT, INSERT, UPDATE, and DELETE commands. Let's look at the syntax of this operator to gain a better understanding of it.

The SQL Syntax

This operator is most frequently used in conjunction with the SELECT command. The following is the syntax:

```
SELECT column_names
FROM table_name
WHERE column_name BETWEEN range_start AND range_end;
```

In Operator

You can define several values in a WHERE clause by using the IN operator. The numerous OR conditions are abbreviated as the IN operator.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

JOIN (INNER, LEFT, RIGHT AND FULL JOINS)

Our database contains vast amounts of data saved in thousands of tables. When we need to look into a specific group of data that meets multiple criteria, we run into issues when combining a huge number of data.

Because the data is dispersed over different tables, we use various joins to combine the tables into a single table to extract the data. SQL join commands allow us to merge the rows, columns, and sections of many tables and see them as a single entity. We may quickly integrate and present data into a single table by using joins.

This makes it simple for us to run queries and transactions on the data we require. Joins are performed based on one or more common fields in two or more tables. They are commonly utilised when a user attempts to extract data from tables that have one-to-many or many-to-many relationships. Now that we have understood what joins are, let's look at the many forms of joins.

Different Types of Joins

Four types of joins that one should be aware of which are (Figure 2.2):

- Right join
- Inner join
- Full join
- Left join

Please see the image below.

How to Determine Which SQL Join to Use

Let us investigate each of them. I'll use the following three tables to demonstrate how to conduct Join operations on them for a better grasp of this topic.

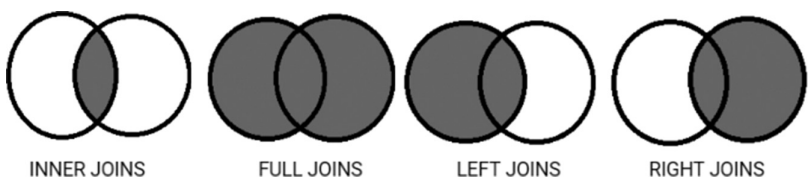


FIGURE 2.2 Types of joins in SQL.

Employee Table:

Emp_ID	EmpF_name	EmpL_name	Age	Email ID	Phone No.	Address
1	Vardha	Kumar	22	vard@abc.com	9889543210	Delhi
2	Hema	Sharma	32	hema@abc.com	9989554422	Mumbai
3	Aayush	Shreshth	24	aayush@abc.com	9977222121	Kolkata
4	Hemant	Sharma	25	hemant@abc.com	9878945666	Bengaluru
5	Swati	Kapoor	26	swati@abc.com	9544567875	Hyderabad

Project Table:

Project ID	Emp ID	Client ID	Project Name	Project Start Date
111	1	3	Project1	2022-04-21
222	2	1	Project2	2022-02-12
333	3	5	Project3	2022-01-10
444	3	2	Project4	2022-04-16
555	5	4	Project5	2022-05-23
666	9	1	Project6	2022-01-12
777	7	2	Project7	2022-07-25
888	8	3	Project8	2022-08-20

Client Table:

C_ID	CF_name	CL_name	Age	C_Email	PhoneNo	Address	Emp_ID
101	Sus	Smit	30	sus@adn.com	9789511231	Kolkata	3
225	Moi	Alia	27	moi@jsq.com	9889543561	Kolkata	3
356	Sohil	Pank	22	so@wja.com	9966332000	Delhi	1
478	Zai	Dagin	40	zai@qkq.com	9955889522	Hyderabad	5
558	Bhas	Paiska	32	bhas@xyz.com	9638953269	Mumbai	2

Inner Joining

This type of join delivers records that have values in both tables that match. As a result, if you run an INNER join operation between the Employee and Projects databases, all tuples with matching values in both tables will be returned as output.

Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.
Column1,....
FROM Table1
INNER JOIN Table2
ON Table1.MatchingColumnName = Table2.
MatchingColumnName;
```

Example:

```
SELECT Emp.EmpID, Emp.EmpF_name, Emp.EmpL_name,
Proj.P_ID, Proj.P_Name
FROM Emp
INNER JOIN Projects ON Employee.EmpID=Projects.EmpID;
```

Output:

EmpID	EmpF_name	EmpL_name	P_ID	P_Name
112	Vardha	Kumar	111	Project1
215	Hema	Sharma	222	Project2
325	Aayush	Shreshtha	333	Project3
385	Aayush	Shreshtha	444	Project4
598	Swati	Kapoor	555	Project5

Full Joining

The Full Join, also known as the Full Outer Join, retrieves all entries that have a match in either the left (Table1) or right (Table2) table.

Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.
Column1,....
FROM Table1
FULL JOIN Table2
ON Table1.MatchingColumnName = Table2.
MatchingColumnName;
```

Example:

```
SELECT Emp.EmpF_name, Emp.EmpL_name, Proj.P_ID
FROM Emp
FULL JOIN Proj
ON Emp.EmpID = Proj.EmpID;
```

Output:

EmpFname	EmpLname	ProjectID
Vardha	Kumar	111
Hema	Sharma	222
Aayush	Shreshtha	333
Aayush	Shreshtha	444
Hemantha	Sharma	NULL
Swati	Kapoor	555
NULL	NULL	666
NULL	NULL	777
NULL	NULL	888

Join on the Left

The LEFT JOIN or LEFT OUTER JOIN returns all records from the left table as well as those that satisfy criteria in the right table. Furthermore, if there are no matching entries in the correct table, the output or result-set will contain NULL values.

Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.
Column1,....
FROM Table1
LEFT JOIN Table2
ON Table1.MatchingColumnName = Table2.
MatchingColumnName;
```

Example:

```
SELECT Emp.EmpF_name, Emp.EmpL_name, Proj.P_ID,
Proj.P_Name
FROM Emp
LEFT JOIN
ON Emp.EmpID = Proj.EmpID ;
```

Output:

EmpF_name	EmpL_name	P_ID	P_Name
Vardha	Kumar	111	Project1
Hema	Sharma	222	Project2
Aayush	Shreshtha	333	Project3
Aayush	Shreshtha	444	Project4
Swati	Kapoor	555	Project5
Hemanthi	Sharma	NULL	NULL

Right Joining

The RIGHT JOIN or RIGHT OUTER JOIN returns all records from the right table as well as those that satisfy a condition in the left table. Furthermore, for records with no matching values in the left table, the output or result-set will contain NULL values.

Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.
Column1,....
FROM Table1
RIGHT JOIN Table2
ON Table1.MatchingColumnName = Table2.MatchingColumnName;
```

Example:

```
SELECT Emp.EmpFname, Emp.EmpLname, Proj.P_ID,
Proj.P_Name
FROM Emp
RIGHT JOIN
ON Emp.EmpID = Proj.EmpID;
```

Output:

EmpF_name	EmpL_name	P_ID	P_Name
Vardha	Kumar	111	Project1
Hema	Sharma	222	Project2
Aayush	Shreshtha	333	Project3
Aayush	Shreshtha	444	Project4
Swati	Kapoor	555	Project5
NULL	NULL	666	Project6
NULL	NULL	777	Project7
NULL	NULL	888	Project8

SQL CONSTRAINT CHECK

The check constraints are the rule or group of rules that aid in the verification of inserted (or updated) data values to tables depending on a certain circumstance. As a result, we may test freshly inserted data values against a predefined rule before allowing them into the table. The primary benefit of check constraints is that ensure all data in a column contains validated values based on the check constraint rule. The CHECK constraint limits the value range that can be entered into a column. When we specify a CHECK constraint on a column, we are saying, it restricts the values that can be assigned to that column. A CHECK constraint on a table might limit the values in some columns based on the values in other columns in the row.

Syntax:

```
CREATE TABLE pets(
        ID INT NOT NULL,
        Name VARCHAR(30) NOT NULL,
        Breed VARCHAR(20) NOT NULL,
        Age INT,
        GENDER VARCHAR(9),
        PRIMARY KEY(ID),
        check(GENDER in ('Male', 'Female', 'Unknown'))
        );
```

The check constraint in the preceding SQL query limits the GENDER to only the categories supplied. The associated database update is cancelled if a new tuple is introduced or an existing tuple in the relation is altered with a GENDER that does not belong to any of the three categories stated.

Query:

From the table mentioned above, the condition is where Students must be over the age of >=17 to enrol in a university.

Student database schema at a university permitted to enrol in a university.

University student database schema is as follows:

```
CREATE TABLE student(
        StudentID INT NOT NULL,
        Name VARCHAR(30) NOT NULL,
        Age INT NOT NULL,
        GENDER VARCHAR(9),
```

```
PRIMARY KEY(ID),
check(Age >= 17)
);
```

Student Relation:

StudentID	Name	Age	Gender
1001	Ron	18	Male
1002	Sam	17	Male
1003	Georgia	17	Female
1004	Erik	19	Unknown
1005	Christine	17	Female

According to the constraint indicated in the check statement in the relation's schema, the age of all students in the above relation is more than or equal to 17 years. However, if the following SQL command is executed:

```
INSERT INTO student(STUDENTID, NAME, AGE, GENDER)
VALUES (1006, 'Emma', 16, 'Female');
```

There will be no database updates, and the age is 17 years.

Check constraint application possibilities entail as follows:

- Alter: Check constraint can also be introduced to an existing relation with the syntax:

```
alter table TABLE_NAME modify COLUMN_NAME
check(Predicate);
```

- Giving a variable name to the constraint check: The following syntax can be used to name check constraints:

```
alter table TABLE_NAME add constraint CHECK_CONST
check (Predicate);
```

- Remove check constraint: A check constraint can be removed from a database relation using the syntax:

```
alter table TABLE_NAME drop constraint
CHECK_CONSTRAINT_NAME;
```

- Drop check constraint: In MySQL, a check constraint can be removed from a relation by using the syntax:

```
alter table TABLE_NAME drop check CHECK_CONSTRAINT_NAME;
```

SUMMARY

SQL clauses and operators were created to assist programmers who are newbies with Structured Query Language in learning and applying the language in their daily job. This chapter gives a brief demonstrates about use of various SQL clauses, SQL commands, SQL statements, and SQL operators. This chapter gives an overview about the SQL clauses and commands like SELECT, INSERT, UPDATE, DELETE, WHERE, JOIN, DISTINCT, ORDER BY, GROUP BY, HAVING, and UNION. The basic objective of this chapter is to provide a very brief introduction you to this powerful language and its clauses and lay the groundwork for you to continue your SQL learning.

CHAPTER 3

SQL Injections

IN THIS CHAPTER

- What is SQLi
- Goals, types, and its mechanism
- Detection and prevention of SQL injection attacks
- Sqlmap and its features
- Prepared statements

The previous chapter taught us about the various SQL statements, clauses, and related commands. In this chapter, we will be learning about a frequent attack vector that involves backend database modification with malicious SQL code to gain access to information that was not intended to be displayed called as SQL injection (SQLi). So, we are moving on with the brief introduction about SQLi and then go on to its mode of operation with examples.

WHAT IS SQL INJECTION (SQLi)?

SQL Injection (SQLi) is an injection attack that allows malicious SQL statements to be executed. These commands are used to control the database server that is connected to the web application. Its flaws can be used by attackers to bypass application security. They can bypass the authentication and authorization of a website or application to obtain the complete contents of a SQL database. It can also be used to create, modify, and delete

DOI: 10.1201/9781003358435-3

database records. Any website or web application that uses an SQL database such as MySQL, Oracle, SQL Server or others can be vulnerable to SQL Injection. It could be used by criminals to illegally access your sensitive data such as customer information, personal data, secrets, intellectual property and more. One of the oldest, most common and deadliest online application vulnerabilities is SQL Injection.

SQL injection can have a high impact on a company bottom line. A successful attack can result in an attacker reading lists of illegal users, deleting entire tables, and in some situations gaining administrator rights to the database, all of which are extremely damaging to a company. Consider the client's loss of trust in your SQLi cost estimate if personal information such as phone numbers, addresses, and credit card information is compromised. Although any SQL database can be attacked using the vector, websites are the most popular targets. As evidenced by their ranking in OWASP's top 10 online application security risks list, one of the most widespread types of security attacks is SQL injection.[1] With the availability of automated SQL injection tools, the risk of exploiting SQLi has expanded, as has the damage it can cause. In the past, attackers had to perform these attacks manually, as the chances of targeting a company using SQL injection were very limited.

GOALS OF SQLi

Following are the goals attained by SQLi:

- **To Extract Data:** The attacker will seize sensitive information. If the admin database is compromised, the entire database is at risk.
- **To Gain Access to Data:** They attempt to circumvent security and gain access to the full database in order to modify the data.
- **Fingerprinting the Database:** In this approach, the attacker will determine the database version and type. This technique allows them to experiment with various types of queries in various applications.
- **Injectable Parameters are Discovered:** susceptible parameters will be discovered for assault using some of the automated technologies.
- **Authentication Bypass:** To gain access to the database, application authentication methods will be circumvented.

- **Database Schema Identification:** To successfully gather information, the database table name, data type of each field, column name, and so on will be acquired from the database table name.
- **To Cause a Denial of Service:** This includes dropping tables and shutting down the system. The attacker attempts to get access to the system in order to carry out a specified command within the database.

MECHANISM OF SQL INJECTION ATTACK

A SQL query is a request to perform a certain action on an application database. Queries can also be used to execute commands on the operating system. When a user executes a query, a set of parameters ensures that only the desired records are returned. Attackers take advantage of this during a SQL injection by introducing malicious code into the query's input form. The first step in a SQL injection attack is to research how the database in question works. This is accomplished by inserting a variety of random values into the query and watching the server's response. After that, attackers utilise what they've learned about the database to create a query that the server will interpret and execute as a SQL command. A database might, for example, store information about customers who have made purchases and have customer ID numbers. An attacker might type "C_ID=1,045 OR 2=2" into the input field instead of searching for a specific customer ID. The SQL query would return all available customer IDs and any relevant data because the statement 2=2 is always true. The intruder now can obtain administrative privileges by avoiding verification. In addition to offering fake access, SQL exploits could be programmed to delete an entire database, get around password requirements, delete records, or add undesirable data.

SQL is a query language developed for managing data in relational databases. It allows you to see, amend, and delete data. Many web apps and websites use SQL databases to store all of their data. Additionally, you can utilise SQL statements to execute specific operating system functions. As a result, a successful SQLi attack can have catastrophic complications:

- Attackers can use it to reveal the credentials of other users in the database. They can then misuse the identities of these users to impersonate them. It is possible that the impersonated user is a database administrator with full access to the database.

- A database query language called SQL allows data to be selected and output from a database. An SQL Injection vulnerability can give an attacker complete access to database server data.
- Additionally, SQL enables you to update and add data to a database. In a financial application, for instance, an attacker could use SQL Injection to modify balances, invalidate transactions, or transfer funds to their own account.
- SQL can be used to drop tables and delete records from a database. Even if the administrator backs up the database, data destruction may cause application downtime until the database is recovered. Furthermore, backups could not include the most recent data.
- You can use the database server to access the operating system on some database servers. This could be deliberate or unintentional. In this situation, an attacker could start with a SQL Injection and then go on to the internal network behind a firewall.

SQL INJECTION TYPES

The three different kinds of SQL injections are in-band SQLi (Classic), inferential SQLi (Blind), and out-of-band SQLi. The ways that SQL injections access the backend data and the level of damage they can do are categorised.

- **In-band SQLi:** The attacker uses the same line of communication to initiate assaults and acquire information. Because of its simplicity and efficiency, in-band SQLi is one of the most prevalent SQLi attacks. This approach is divided into two sub-variations:
  - **SQLi Based on Error Messages:** The attacker takes steps that cause the database to generate error messages. The data provided by these error messages could be used by the attacker to obtain knowledge about the database's structure.
  - **Union-Based SQLi:** this technique makes use of the UNION SQL operator, which combines many database select statements into a single HTTP response. This response may contain information that the attacker can use.

- **Blind (Inferential) SQLi:** In order to understand the structure of the server, the attacker delivers data payloads to it and then observes its behaviour and reaction. Because data is not communicated from the website database to the attacker, this method is known as blind SQLi. As a result, the attacker is unable to view information about the attack in-band. The Blind SQL injections rely on the server's response and behaviour patterns, thus they're slower to execute but just as dangerous. The various kinds of blind SQL injections include the following:
  - **Boolean:** The attacker sends a SQL query to the database, requesting a response from the application. Depending on whether the query is true or false, the response will differ. The information in the HTTP response will alter or remain unchanged depending on the outcome. The attacker can then determine if the message produced a true or false response.
  - **Time-based:** The attacker sends a SQL query to the database, which causes it to wait (for a specified number of seconds) before responding. The attacker can tell whether a query is valid or false based on how long it takes the database to answer. An HTTP response will be generated immediately or after a waiting period based on the result. Without relying on database data, the attacker can determine if the message they used returned true or false.
- **Out of Band SQLi:** Only if specific functions on the database server used by the web application are enabled, this type of attack is conceivable. Usually, this kind of attack is used in conjunction with in-band and inferential SQLi assaults. When a server is too slow or unreliable to carry out these tasks, or when an attacker is unable to start an attack and gather information over the same channel, SQLi is utilised. To send data to an attacker, these techniques rely on the server's capacity to deliver DNS or HTTP requests.

DETECTION AND PREVENTION OF SQL INJECTION ATTACKS

A successful SQL injection attack could cause serious harm by disclosing private information and eroding customer trust. Detection of this kind of attack must be made as soon as possible for this reason. Web application firewalls are the solution that is most frequently used to stop SQLi

threats (WAFs). WAFs are based on a library of up-to-date attack signatures and can be configured to warn against hazardous SQL queries in online applications.

The following steps can be taken by businesses to prevent a SQL injection attack:

- **Employees Should Be Educated On Preventative Techniques:** It's critical that IT personnel, such as DevOps professionals, system administrators, and software development teams, undergo comprehensive security training to understand how SQLi attacks occur and how to prevent them in web applications.
- **Don't Place Your Faith On User Input:** Input from a user in a SQL query improves the chances of a successful SQL injection. Putting security controls around user input is the best method to mitigate this type of danger.
- **Instead of A Blocklist, use an Allowlist:** Because hackers can usually bypass a blocklist, using an allowlist instead of a blocklist to validate and filter user input is suggested.
- **Make Sure Your Applications are Current and that Your Route is Updated:** Outdated software is one of the most common SQL injection issues. Older technology is less likely to have SQLi security built in, and unpatched software is often easier to manipulate. This also applies to programming languages. Languages and syntax from the past are more vulnerable. Use PDO as a replacement for older MySQL, for example.
- **Make use of Tried-and-True Preventative Techniques:** Insufficient protection against a SQLi attack is provided by query strings built from scratch. Input validation, prepared statements, and parameterised queries are the best ways to safeguard web applications.
- **Regularly Scan for Security Threats:** Scanning web applications on a regular basis will detect and fix potential vulnerabilities before they do major harm.

Simple SQLi Example

The first example is extremely straightforward. It demonstrates how an attacker can bypass application security and authenticate as the

administrator by exploiting a SQL Injection vulnerability. The script below is pseudocode that runs on a web server. It is a simple example of using a *username* and *password* to authenticate. The *users* table in the sample database has the following columns: username and password.

```
# Define POST variables

username = request.POST['u_name']
password = request.POST['p_word']

# SQL query vulnerable to SQLi

SQL = "SELECT C_id FROM users WHERE username='" +
username + "' AND password='" + password + "'"

# Execute the SQL statement

database.execute(SQL)
```

SQL Injection is possible in these input fields. An attacker could utilise SQL commands in the input to change the SQL statement that the database server executes. They could, for example, use a single quote trick to set the `password` field to:

```
Password' OR 2=2
```

As a consequence, the database server runs the following SQL query:

```
SELECT C_id FROM users WHERE username='username' AND
password='password' OR 2=2'
```

No matter what the username or password is, the `WHERE` clause always returns the first id from the `users` table because of the OR `2=2` statement. The administrator is frequently the first user `id` in a database. The attacker not only gets administrator rights but also bypasses authentication. To have additional control over how the query is executed, they can also comment out the remaining SQL statement:

```
-- MySQL, MSSQL, Oracle, PostgreSQL, SQLite
' OR '2'='2' --
' OR '2'='2' /*
```

```
-- MySQL
' OR '2'='2' #
-- Access (using null characters)
' OR '2'='2' %00
' OR '2'='2' %16
```

SQLMAP: TEST A WEBSITE SQL INJECTION VULNERABILITY

The risk of cyberattacks is rising as technology advances. For a variety of reasons, websites are growing more vulnerable to these attacks. SQL injection attacks are on the rise, putting web application security at risk. Scanning your online application can help you avoid data loss, cyberattacks, and business disruption. As a web developer, you should constantly take precautions to safeguard your web application against SQL vulnerabilities that hackers can exploit. The sqlmap utility is a popular tool for identifying and exploiting SQL injection threats.

SQLMAP

Sqlmap is a free and open source penetration testing tool for detecting and leveraging SQL injection vulnerabilities and gaining control of database systems. It comes with a powerful detection engine, a slew of specialised features for the ultimate penetration tester, and a slew of switches for database fingerprinting, data extraction from databases, access to the underlying file system, and out-of-band command execution on the operating system.

Features of Sqlmap

- It provides full support for MySQL, Firebird, PostgreSQL, Microsoft Access, Oracle, IBM DB2, SQLite, Sybase, SAP MaxDB, Informix, MariaDB, MemSQL, TiDB, CockroachDB, Microsoft SQL Server, H2, MonetDB, Apache Derby, Amazon Redshift, Vertica, Mckoi, HSQLDB, Altibase, MimerSQL, CrateDB, Greenplum, Cubrid, Presto database system.
- Full Boolean-based blind, time-based blind, error-based, query-based, stacked queries, and out-of-band SQL injection capabilities.
- By giving DBMS credentials, IP address, port, and database name, it is possible to connect to the database directly without using SQL injection.

- It is possible to list down the users, password hashes, privileges, roles, databases, tables, and columns.
- It is possible to crack password hash types using a dictionary-based attack because they are automatically recognised.
- According to the user's preferences, there is support for dumping entire database tables, a selection of records, or certain fields. From each column entry a subset of characters can also be chosen by the user to be unloaded.
- Support for searching across all databases for specified database names, specific tables, or specific columns. For instance, it may be used to search tables holding custom application credentials where the appropriate columns' names contain strings such as name and pass.
- When using MySQL, PostgreSQL, or Microsoft SQL Server, you can download and upload any file from the database server's underlying file system.
- When using MySQL, PostgreSQL, or Microsoft SQL Server as the database software, support is given for running arbitrary commands on the database server's underlying operating system and receiving its standard output.
- Support for establishing an out-of-band stateful TCP connection between the database server's operating system and the attacker workstation. This channel may be an interactive command prompt, a Meterpreter session, or a graphical user interface (VNC) session, depending on the user's preferences.
- Support for user privilege escalation in database operations using the Meterpreter getsystem command of Metasploit.

The detailed description of all the features can be learned from wiki[2] (Figure 3.1).

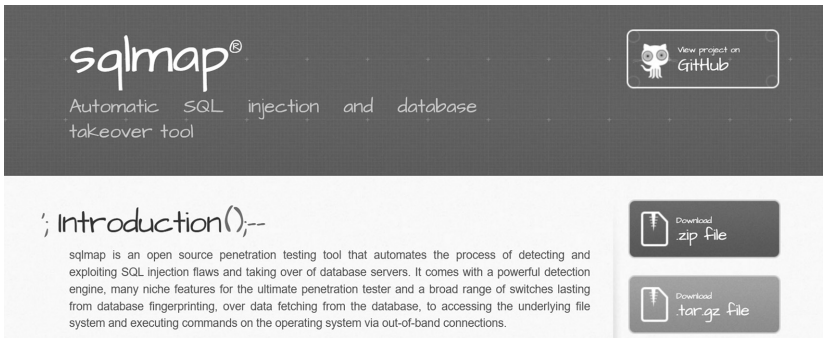


FIGURE 3.1 Download link for Sqlmap.

How to Download

You can get the most recent zip ball or tarball here. The best way to get sqlmap is to clone the Git repository:

```
git clone --depth 1 https://github.com/sqlmapproject/
sqlmap.git sqlmap-dev
```

WHERE SQLMAP MAY BE USED

If you see a web address that ends in ".php?id=", the website may be vulnerable to SQL injection in this mode, and an attacker may be able to access information in the database.

To assess security weaknesses in targeted online applications, Sqlmap accepts `GET` and `POST` protocols. If the URL has a bold `GET` parameter, the website is more vulnerable to SQL injection. Hackers will be able to access information in the database this way. By substituting the GET parameter with an asterisk (*), we can check the website's vulnerability.

For instance, go to http://checkphp.vulnerableweb.com/artists22.php?artist=4*.

If this URL returns an error, the site is allowed. You can use sqlmap to scan or exploit web vulnerabilities. Here, we learn how to use sql map to check for SQL injection vulnerabilities on the web. For ".php? id=" to "/", developers use a URL rewrite rule in .htaccess. It doesn't mean it's safe.

DAMN VULNERABLE WEB APPLICATION (DVWA)

The Damn Insecure Online Application is a highly vulnerable PHP/MySQL web application. Its primary goal is to help security professionals test their skills and tools in a legal environment, help web developers better understand web application security processes, and help students and

teachers learn about web application security in a controlled classroom environment.

The goal of DVWA is to practice some of the common web vulnerabilities in a basic, straightforward interface with various levels of difficulty. Please be aware that this program has both documented and undocumented vulnerabilities. This is done on purpose. It is recommended to research as many issues as possible.

We will use DVWA as a demo environment to test our sqlmap commands in this section, but first, we need to install the following software:

- Sqlmap install it using following code : `$ sudo apt-get install sqlmap`
- DWV

  The procedure is simple; simply follow the guide to install DVWA on your system.[3]

Pass `-h` to `sqlmap` to display the command's help menu and verify that it is installed.

```
$ sqlmap –h
```

If you don't want to install DVWA, you can use this publicly accessible vulnerable website instead:

http://checkphp.vulnerableweb.com/artists22.php?artist=4*.

Let us get started now that everything is set up.

Determine the Database Management System (DBMS) in the Site

I want you to login to DVWA using admin as username and password as password, then go to DVWA Security in the bottom left and set the Security level to Low, which will allow us to exploit the website in its most vulnerable state (Figure 3.2).

Returning to the hacker terminal, the following command is used to determine the database list for this website:

```
$ sqlmap -u "http://localhost/vulnerabilities/
sqli/?id=1&Submit=Submit" --cookie "PHPSESSID=u8e7b7vb
kkienkafe68a6pabzf; security=low" –dbs
```

First, we use the `-u` parameter to specify our target URL, which in my case is `localhost/`, but you should specify where your DVWA is installed,

FIGURE 3.2 DVWA login page.

for example, if it's on another machine in the same network and in the DVWA folder.

192.168.1.3/DVWA/vulnerabilities/sqli/?id=1&Submit=Submit.

We also pass the —cookie argument, because DVWA requires login to begin performing SQL injection, so simply passing our cookie will log us in. You can find your cookie by going to Developer Console>Network, looking for any request, scrolling down to the Request Headers section, and looking for Cookie.

We use –DBS to retrieve a list of the website's available databases; the following is the result:

```
available databases [2]:
[*] dvwadb
[*] information_schema
```

Listing of Tables in a Database

If you want to search a certain database, use the -D parameter to indicate the database name, and then use the - table parameter to display a list of all tables in that database:

```
$ sqlmap -u "http://localhost/vulnerabilities/
sqli/?id=1&Submit=Submit" --cookie "PHPSESSID=u8e7b7vb
kkienkafe68a6pabzf; security=low" -D dvwadb –tables
```

Output:
Database: dvwadb
[2 tables]

Guestbook
Customers

Great, we have two tables in this database; we will dump the Customers table in the next part.

Getting Rid of a Table

We must give the database, the table, and the - columns parameter to examine the available columns of a certain table:

```
$ sqlmap -u "http://localhost/vulnerabilities/
sqli/?id=1&Submit=Submit" --cookie "PHPSESSID=u8e7b7vb
kkienkafe68a6pabzf; security=low" -D dvwadb -T
customers –columns
```

Here's what happens when we use the -T argument to define the table name:

Output:
Database: dvwadb
Table: Customer

Column	Type
password	varchar(32)
user	varchar(15)
avatar	varchar(70)
failed_login	int(3)
first_name	varchar(15)
last_login	timestamp
last_name	varchar(15)
user_id	int(6)

Let's dump this table so we can see all of its rows:

```
$ sqlmap -u "http://localhost/vulnerabilities/
sqli/?id=1&Submit=Submit" --cookie "PHPSESSID=u8e7b7vb
kkienkafe68a6pabzf; security=low" -D dvwadb -T
Customer –dump
```

We just substituted - dump for - columns; this will prompt you with various questions, including whether you want to save the hashes to a temporary file or crack the passwords using a dictionary-based attack; I selected Y (yes) for both, and the result.[4]

Mitigating the SQL Injection Attack with Prepared Declarations

When user-supplied data is utilised as part of a SQL query, SQL Injection is a software vulnerability. An attacker can submit a legal SQL statement that affects the logic of the application's initial query due to faulty data validation. As a consequence, the attacker has access to sensitive data belonging to other users, as well as unlawful access to the entire system. SQL Injection vulnerabilities are still common, despite being simple to address. In this section, we'll look at how to avoid these flaws by using good coding methods also we will discuss about prepared statements, how they function, and how to use them.

Prepared Statements

A prepared statement is a parameterised, reusable SQL query that requires the developer to create the SQL command and user-supplied data separately. SQL Injection risks are avoided since the SQL command is executed safely.

In PHP, here's an example of an unsafe approach:

```
$query = "SELECT * FROM users WHERE user = '$user_
name' and password = '$password '";
$result = mysql_query ($query);
```

From the above code it is clearly evident that the data that the user has provided is directly inserted into the SQL query mentioned. If the user pungin or inputs admin and 'a' or '2'='2, the user will be able to access the login credentials of the admin account without even having knowledge of user login pin or password because the SQL statement declared above has been tampered/modified.

Here is an example of a prepared statement approach in PHP:

```
$stmt =  $mysqli->prepare ("SELECT * FROM users WHERE
username =? AND password =?");
$stmt->Jheeta_Lal ("jl", $username, $password);
$stmt->execute();
```

In this example, the user-supplied data is not directly incorporated into the SQL query. The user's data has been replaced by a symbol. That serves as a stand-in and temporarily replaces the data. The user's data is added afterwards after the SQL query has already been pre-compiled using placeholders. If the user inserts admin and a' or '2'='2, the initial SQL query logic would not be changed. Instead, the database will search for the entries where user admin whose password is literally a' or '2'='2.

Mechanism of Action of Prepared Statements

Before we talk about how the prepared statement works, let's take a look at SQL query processing. It involves six basic steps:

- **Parsing:** The query in SQL is chunked down tokens (Individual words). It will run then run a syntax error and misspelling checks to ensure the validity of the SQL query.
- The Database Management System (DBMS) determines the query's legitimacy using a semantics check. Exist the columns and table that are specified? The major question here is does the user have the necessary rights to run this query?
- **Binding:** Byte code is used to translate the query into a form that computers can interpret. The query is then put together and delivered to the database server to be optimised and run.
- **Query Optimization:** The DBMS selects the most cost-effective algorithm to execute the query.
- **Cache:** The best method is saved in the cache so that the next time the identical query is conducted, the first four steps will be skipped, and the execution will begin immediately.
- **Execution:** The query is carried out, and the user receives the results.

But given that they are distinct from a regular query, how does a prepared statement go through this process?

Although the process is the same, there are a few differences:

Semantics Checking is equivalent to parsing. After the database engine detects the placeholders, binding builds the query with placeholders. Later,

the user's data will be added. The step called 'Cache' doesn't change. The query is cached so it can be used again.

There is still a phase after caching and before execution: placeholder replacement. The user's information is now entered in place of the placeholders. The final query won't go through the compilation process again because the query has already been pre-compiled (Binding). Because of this, the user-provided data will always be treated as a basic string and is unable to alter the logic of the original query. The query will therefore be resistant to SQL Injection flaws for that data. That is a method of taking advantage of a SQL statement's inherent weakness by injecting malicious SQL statements into its entry field for final execution, as described in the chapter about SQLi above. Since 1998, when it initially surfaced, it has mostly targeted stores and bank accounts. It can produce significant effects when combined with other types of attacks like DDOS attacks, cross-site scripting (XSS), or DNS hijacking.

Terminology

- **Validation:** It is a process of checking or confirming if the user input fulfils the criteria predefined viz., string contains no stand alone single quotation marks.
- **Sanitization:** It is the process of modifying the input to ensure that it is valid (such as doubling single quotes); basically, it can be called as validity check.

To prevent SQL injection, all input to be concatenated in dynamic SQL must be properly filtered and sanitised.

Anatomy of a SQL Attack: An SQL attack has the following two parts:

- **Research:** Examine the vulnerable parts of the user application that connect to the database.
- **Attack:** Enter malicious fields that can change the query to your advantage.

Example 1: Consider the following piece of Java code for an authentication form:

```
String query = "SELECT userName, balance FROM accounts"
   + "WHERE userID=" + request.getParameter("userID") +
"and password='" + request.getParameter("Password") + "'";
try
{
   Statement statement = connection.createStatement();
   ResultSet rs = statement.executeQuery(query);
   while (rs.next())
   {
       page.addTableRow(rs.getString("userName"),
                        rs.getFloat("balance"));
   }
}
catch (SQLException e)
   {}
```

When a user inputs his or her UserID and password in normal circumstances, the following statement is generated for execution:

```
SELECT userName, balance
FROM accounts
WHERE userID=510 and password='whatislife'
```

A hypothetical SQL injection attack would take advantage of the password field to construct a boolean expression that would evaluate to true in all situations. Consider the following settings for the userID and password fields:

```
userID = 2' or '2' = '2
password = 2' or '2' = '2
```

The SQL statement is then transformed into

```
SELECT userName, balance
FROM accounts
WHERE userID='2' OR '2'='2' and
      password='2' OR '2'='2'
```

Because the criteria (OR 2=2) is always true, the query will return a value. Without knowing the user's username or password, the system has successfully authenticated the user.

Using a prepared statement to build a parameterised query, the vulnerability can be mitigated as follows:

```
String query = "SELECT userName, balance "+
               "FROM accounts WHERE userID = ?
                    and password = ?";
try {
PreparedStatement statement = connection.
prepareStatement(query);
statement.setInt(1, request.getParameter("userID"));
ResultSet rs = statement.executeQuery();
while (rs.next())
{
   page.addTableRow(rs.getString("userName"),
                           rs.getFloat("balance"));
}
} catch (SQLException e)
   { ... }
```

If an attacker tries to set a value for the userID column that isn't a basic integer, `statement.setInt()` will throw a SQLException instead of allowing the query to finish.

Example 2: Consider another form of attack that could occur during authentication:

```
String query = "SELECT u_ID,Name, password Hash"+
             " FROM users WHERE Name = '"
             + request.getParameter("user") + "'";
int userID = -1;
HashMap userGroups = new HashMap();
try
{
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery(query);
rs.first();
userID = rs.getInt("u_ID");

if (!hashOf(request.getParameter("password")).
equals(rs.getString("password_Hash")))
{
```

```
   throw BadLoginException();
}

String userGroupQuery = "SELECT group FROM
groupMembership"+
                            " WHERE u_ID = " + u_ID;

rs = statement.executeQuery(userGroupQuery);

while (rs.next())
{
   userGroup.put(rs.getString("group"), true);
}
}
catch (SQLException e){}
catch (BadLoginException e){}
```

Here's an illustration of a typical inquiry:

```
SELECT userID, userName, passwordHash
FROM users
WHERE userName = 'Anam'
```

The following could be injected into the userName field by the attacker.

```
Anam';
INSERT INTO groupMmbership (userID, group)
VALUES (SELECT userID FROM users
WHERE userName='Anam', 'Administrator'); --
```

As a result, the real question will become:

```
SELECT u_ID, Name, password hash FROM
       users WHERE Name = 'Anam';
INSERT INTO groupMmbership (u_ID, group)
VALUES (SELECT u_ID FROM users
WHERE userName='Anam', 'Administrator'); --'
```

Another SQL statement will be appended to the original statement, resulting in the user being added to the Administrator database. The following example shows how to minimise the attack by utilising a prepared statement with a parameterised query.

```
String query = "SELECT u_ID, Name, passwordHash"+
             " FROM users WHERE userName = ?";

try
{
PreparedStatement statement =
      connection.prepareStatement(userLoginQuery);
statement.setString(1, request.getParameter("user"));
ResultSet rs = statement.executeQuery();
}
```

Consider this third example of a query vulnerability, which is addressed further down:

```
String query = "INSERT INTO users VALUES(" +
      request.getParameter("userName") + ");";
```

The following is a broad question:

```
INSERT INTO users VALUES("Anam")
```

Consider what would happen if the attacker typed the following into the userName field:

```
"Anam); DROP TABLE users;"
```

After that, the inquiry will become:

```
INSERT INTO users VALUES ("Anam"); DROP TABLE users;
```

When this query is run, it entirely deletes the users table. Again, a prepared statement can be used as a workaround.

What is the Benefit of Utilising a Prepared Statement in Java?

The input is 'sanitised' by a prepared sentence. This ensures that whatever the user types are, they are regarded as a string literal in SQL rather than being included in the SQL query. It may also detect and eliminate dangerous code as well as escape particular characters. In other languages, like PHP, the filter input or filter input array can be used to sanitise the string.

What about Sanitization of the Input?

The technique of deleting any undesired characters from user-supplied data (e.g.,'/ ") is known as input sanitization. This strategy, however, is insufficient to avoid all sorts of SQL Injection and is extremely difficult to master. To get around sanitization, there are a variety of bypass strategies available, such as encoding data. As much as possible, avoid utilising input sanitization. Prepared statements or stored procedures should be used instead.

SUMMARY

In this chapter, we learned about SQL Injection, a type of attack that makes use of weak SQL statements. Using SQL injection, it is possible to access, insert, update, and delete data as well as circumvent authentication processes. Use sound security practices while writing SQL statements to reduce the risk of SQL injection. We also talked about the SQL injection tools, including Sqlmap, and we got a quick explanation of prepared statements and input sanitization.

NOTES

1 https://owasp.org/www-project-top-ten/Wasp to ten, Wasp.
2 Feature-Miroslav Stampar, Github.com.
3 How To Setup DVWA Using XAMPP (Windows Tutorial)-Effecthacking.com
4 How to Scan SQL Injection Vulnerable Sites using Sqlmap-Rimsha Ashraf, Root Install.

CHAPTER 4

SQL Functions

IN THIS CHAPTER

- What are Functions in SQL
- Different types of function
- Syntax and Examples

In the previous chapter, we learned about SQLi, how to prevent SQLi attack, and sqlmap and its functions, and we also discussed a brief introduction about prepared statements and their function in SQL. A function is a formula that accepts one or more arguments as input, processes them, and then returns output. It is a programming construct that returns a single value and accepts parameters. Function parameters can change the result or return the result. The beauty of a function is that it can be embedded in an expression because it is self-contained. An expression in Oracle SQL is a statement of SQL code or even another function by definition.

SQL FUNCTIONS

SQL functions are a very powerful feature that can be used to perform calculations on data, change individual data items, change output for groups of rows, format data and numbers for display, and convert column data types. An SQL function always outputs a value and can accept input.

SQL functions are regularly used routines that help with database processing, modification, and maintenance. SQL functions are simple

DOI: 10.1201/9781003358435-4

subroutines that are often used and reused in SQL database systems to process and manipulate data. In order to create and manage databases, all SQL database systems include DDL (Data Definition Language) and DML (Data Manipulation Language) tools.

There are five general categories of functions in SQL. Object reference functions that provide access to unique object pointers, aggregation functions that group rows into groups, analytic functions that also group but allow in-depth data analysis, and user-defined functions that you can create using a programming language such as PL/SQL, are other types of functions.

- **Single-Row Function:** Individual rows of a query can be operated on individually. In other words, the same process can be performed for each row that the query retrieves using the single-row function.
- **Aggregation Functions:** These functions combine repeating groupings into a set of rows and aggregate repeating values into things like sums or averages.
- **Analytical Functions:** Summaries of subsets of aggregates are produced by analytics, not aggregates that break iterations into distinct parts.
- **Object Reference Functions:** These functions refer to values using pointers. Object reference functions typically dereference values from objects or refer to other objects.
- **User-Defined Functions:** PL/SQL can be used to create custom functions, extending Oracle SQL's extensive built-in capability library.

Aggregate functions, scalar (non-aggregate) functions, and analytical functions are major forms of SQL functions, which are discussed in detail in this chapter. Non-aggregate functions work with GROUP BY and operate on each record individually, whereas aggregate functions operate on several records and produce a summary. SQL has a plethora of built-in functions for performing various data calculations.

Aggregate Functions

It returns a single value after performing a calculation on a set of values. They can be used in a SELECT statement's select list or the

HAVING clause. To calculate the aggregation on categories of data, use an aggregation in conjunction with the GROUP BY clause. To calculate the aggregation on a given range of values, use the OVER clause. The GROUPING or GROUPING _ID aggregations cannot be followed by the OVER clause.

When executed on the same input data, all aggregate functions are deterministic, which means they always return the same output.

Function	Description
SUM()	This returns the sum of a group of values.
COUNT()	It returns the number of rows either based on a condition, or without a condition.
AVG()	Returns the average value of a column (numeric).
MIN()	Returns the minimum value of a column.
MAX()	Returns a maximum value of a column.
FIRST()	Return the first value of the column.
LAST()	Returns the last value of the column.

Analytic Functions

Analytic functions calculate a total value from a set of rows. Analytic functions, unlike aggregate functions, can return numerous rows for each group. Within a group, analytic functions can be used to generate moving averages, running totals, percentages, or top-N outcomes.

Scalar SQL Functions

The Scalar Functions in SQL are used to return a single value from the given input value. Here are some of the most popular aggregate functions. Let us look into each one of the above functions in depth.

Function	Description
LCASE()	convert string column values to lowercase
UCASE()	It converts a string column values to uppercase.
LEN()	It returns the length of the text values in the column.
MID()	It extracts substrings from column values having string data type.
ROUND()	It rounds off a numeric value to the nearest integer.
NOW()	It returns the current system date and time.
FORMAT()	Format how a field must be displayed.

SQL Server Mathematical Functions

Capabilities are objects that have a bunch of SQL proclamations. Each capability acknowledges boundaries as information and plays out a succession of tasks prior to bringing output back. A solitary outcome set is returned by a capability. SQL Server gives a few numerical capabilities to perform essential numerical estimations. SQL has numerous numerical capabilities that permit you to perform business and designing estimations. Not all the SQL math capabilities are utilised in normal everyday activities.

For instance, we can use these functions to find the square root, logarithmic, round, floor, elementary exponential value, and trigonometric functions. However, there are several commonly used functions cited below:

Name	Description and Syntax
ABS	Returns the absolute value Syntax: `ABS (number|expression)` Example: Select abs(-2); Returns 2
ACOS	Returns the arc cosine of an argument Syntax: ACOS(numeric_expression) Example: Select acos(0); Returns 1.57079632679
ASIN	Returns the arc sine of an argument Syntax: ASIN(numeric_expression) Example: SELECT ASIN(0.5) angle_1_radians, Returns 0.523598775598299
ATAN	Returns the arc tangent of an argument Syntax: ATAN(x) Example: SELECT ATAN(1); Returns 0.7855398163
CEILING,CEIL	It floats values are rounded to the nearest integer value. Syntax: CEIL(numeric_expression) Example: SELECT CEIL(50.49); Returns 61
COS	Returns the cosine of an argument Syntax: COS(numeric_expression) Example: SELECT COS(0) cos_zero; Returns 1

(*Continued*)

Name	Description and Syntax
COT	Returns the cotangent of an argument Syntax: COT(numeric_expression) Example: SELECT COT(PI()/5) cot_one_fifth_pi; Returns 11.4300523
EXP	Returns the value of the e constant (2.71828...) raised to a certain number's power. Syntax: EXP(number | expression) Example: SELECT EXP(5); Returns 148.413159103
LN	Returns the argument's natural logarithm.
LOG	Returns the first argument's natural logarithm.
LOG10	It returns the argument's base-10 logarithm.
LOG2	It returns the argument's base-2 logarithm.
MOD	It returns the leftover part (modulo) after dividing a number by another. Syntax: MOD(a,b); Example: SELECT 20/5 as integer, MOD (20,5) as remainder; Returns 20, 5
PI	The result is pi, which is 3.14159265358979. Syntax : SELECT PI()
POWER	POWER returns a value that has been raised to the power of a given number. Syntax: POWER(numeric_expression,power) Example: SELECT POWER(5,1); Returns 5
RAND	Returns a random floating-point value Syntax: SELECT RAND();
ROUND	Rounds a number to a specific precision Syntax: SELECT ROUND(..value.., number_of_decimal_places)
SIGN	Returns the sign of an argument Syntax:
SIN	Returns the sine of an argument Syntax: SIN(numeric_expression)
SQRT	Returns the square root of an argument Syntax: SQRT(number | expression)
TAN	Returns the tangent of an argument Syntax: tan A=a/b
TRUNCATE	Truncates to a specified number of decimal places Syntax: TRUNCATE(n, d)

CONVERSION FUNCTION

The Oracle built-in function library includes type conversion functions in addition to SQL utility functions. In some cases, the query may expect input of a certain data type but receives it in a different data type. In some circumstances, Oracle tries implicitly to transform the unexpected value to a compatible data type that can be substituted in place without compromising application continuity. Oracle can convert types intuitively or explicitly, depending on the programmer's preference (Figure 4.1).

Explicit vs. Implicit

Practically, all programming languages provide processes or facilities for data type conversion, which is the process of changing the data type of a value into a different data type. When we shift our focus to SQL Server in order to discuss the specifics of SQL data conversion procedures, we can divide the data conversion process into two categories: implicit and explicit conversions. SQL Server performs implicit conversion for internal purposes, and you can learn more about it in the later section of Server Implicit Conversion. Explicit conversion is done explicitly by a database programmer or administrator, and it is done with the help of any data conversion function at the same time.

Conversion of Implicit Data Types

Oracle can implicitly convert a VARCHAR2 or CHAR value to a NUMBER or DATE type value. Similarly, an Oracle server can convert a NUMBER or DATA type value to character data automatically. It's worth noting that the implicit interconversion only occurs when the character represents a valid numeric or date type value.

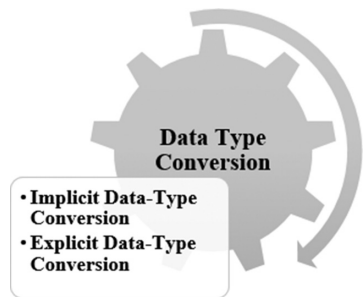


FIGURE 4.1 Types of conversions.

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
DATE	VARCHAR2
NUMBER	VARCHAR

Example: Take a look at the SELECT queries below. Because Oracle internally interprets 2000 and '2000' as the same, both queries will return the same answer.

Query 1:

```
SELECT emp_id, F_name, Bonas
FROM employees Data
WHERE bonas > '20000';
```

Query 2:

```
SELECT emp_id, F_name, Bonas
FROM employees Data
WHERE bonas > '20000';
```

Output:

Emp_ID	F_Name	Bonas
10524	Stevan	34,000
10785	Naive	27,000
10289	Lexi	30,000

SQL Conversion of Explicit Data Types Conversion

It is functions are single-row functions that can typecast the value of a column, a literal, or an expression. The three functions that conduct data type cross-modification are TO CHAR, TO NUMBER, and TO DATE.

Syntax:

```
TO_CHAR (number1, [format], [nls_parameter])
```

Using the TO CHAR Function with Dates

Syntax:

```
TO_CHAR (date, 'format_model')
```

The format model is as follows:

- It is case sensitive and must be contained in single quote marks.
- Any valid date format element can be used.
- Has an FM element to suppress leading zeros or erase padded blanks.
- A comma separates it from the date value.

Example:

```
SELECT Emp_id, TO_CHAR (J_date, 'MM/YY') Month_Hired
FROM employees data
WHERE L_name = 'DEPP';
```

Using the TO CHAR Function with Numbers

Syntax:

```
 TO_CHAR (number, 'format_model')
```

To show a number value as a character, you can use the TO CHAR function with the following format elements:

Example:

```
SELECT TO_CHAR(salary, '$55,555.00') SALARY
FROM employees data
WHERE L_name = 'TUD';
```

Output:

Salary
$9,000

The to Number and to Date Functions
Using the TO NUMBER function, convert a character string to a numeric format:

Syntax:

```
TO_NUMBER(char[, 'format_model'])
```

Using the TO DATE function, convert a character string to a date format:

```
TO_DATE(char[, 'format_model'])
```

There is a fx modifier on these functions. This modification provides the exact match for a TO DATE function's character parameter and date format model.

Example:

```
SELECT L_name, J_date
FROM employees data
WHERE J_date = TO_DATE ('JUNE 24, 1995', 'fx Month DD,
YYYY');
```

Output:

Last Name	Hire Date
True	24-June-95

The conversion function lets us to alter date formats for display, convert column datatypes, perform calculations on data, modify individual data items, and manipulate output for groups of rows

GENERAL FUNCTIONS IN SQL

The functions are used to handle null values and deal with any data type. Numerous mathematical functions, component manipulation functions, including virtual components, Date Time functions, forward and reverse geocoding, interactive selection, and system functions such as reporting GPU types, table caching, manual thread configuration, value sequences, and vector numeric functions are all included in these general-purpose functions (Figure 4.2).

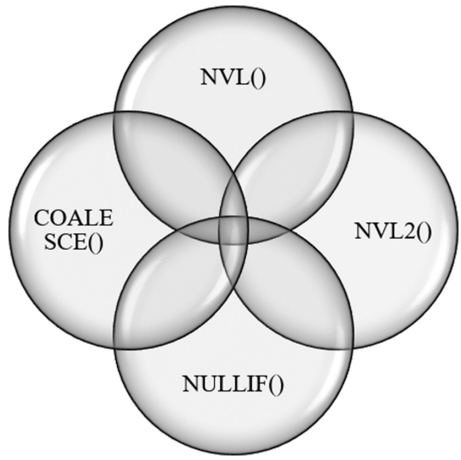


FIGURE 4.2 General function in SQL.

NVL()

It is a SQL function that converts a null value to a valid value. Date, letter, and integer are examples of data types that can be employed. The data types must be compatible, i.e., expr1 and expr2 must be of the same data type.

Syntax:

```
NVL (expr1, expr2);
```

expr1 is the source worth or articulation that is equipped for containing an invalid.

expr2 is the desired outcome for converting the null.

NVL2 Function

The NVL2 capability assesses the underlying articulation (expr1, expr2, expr3). The NVL2 capability returns the subsequent articulation on the off chance that the principal articulation isn't invalid. In the event that the primary articulation is invalid, the third articulation is returned, for example, NVL2 returns expr2, assuming expr1 isn't invalid. NVL2 returns expr3 on the off chance that expr1 is invalid. Any information type can be utilised in the expr1 contention.

Syntax:

```
NVL2 (expr1, expr2, expr3)
```

expr1 is the source value or expression, which may or may not contain a null value.

If expr1 is not null, the value returned is expr2.

If expr1 is null, the value returned is expr3.

DECODE()

It performs the duties of an IF-THEN-ELSE or CASE statement, simplifying conditional searches. With the help of IF-THEN-ELSE logic, which is ubiquitous in computer languages, the DECODE function decodes an expression. The DECODE function decodes the expression after comparing each search value to it. If the expression matches the search, the result is returned.

When a search value does not match any of the result values and the default value is omitted, a null value is returned.

Syntax:

```
DECODE(col|expression, search1, result1
 [, search2, result2,...,][, default])
```

COALESCE()

It really takes a look at the main articulation, and in the event that the principal articulation isn't invalid, it returns that articulation; otherwise, it does a COALESCE of the excess articulations.

The benefit of the COALESCE() capability over the NVL() capability is that the COALESCE capability can take different substitute qualities. In basic words, COALESCE() capability returns the main non-invalid articulation in the rundown.

Syntax (Sentence Structure):

```
COALESCE (expr_1, expr_2, ... expr_n)
```

NULLIF()

The NULLIF capability looks at two articulations. Assuming that they are equivalent, the capability brings invalid back. In the event that they are not equivalent, the capability returns the principal articulation. You can't indicate the strict NULL for first articulation.

Syntax (Sentence Structure):

```
NULLIF (expr_1, expr_2)
```

LVL()

LVL assesses a condition when one of the two operands of the condition might be invalid. The capability can be utilised exclusively in the WHERE provision of a question. It takes as a contention a condition and returns TRUE in the event that the condition is FALSE or UNKNOWN and FALSE assuming the condition is TRUE.

Syntax (Sentence Structure):

```
LVL( condition(s) )
```

CONDITIONAL STATEMENTS IN SQL

They are utilised to characterise what rationale is to be executed in view of the situation with some condition being fulfilled. There are two sorts of contingent articulations upheld in SQL strategies:

- CASE
- IF

The CASE explanation goes through conditions and returns a worth when the primary condition is met (like an on the off chance that else proclamation). In this way, when a condition is valid, it will quit perusing and return the outcome. Expecting no conditions are substantial, it returns the value in the ELSE statement. If there is no ELSE condition and no circumstances are authentic, it brings NULL back

Syntax (Sentence Structure):

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN condition THEN resultN
    ELSE result
END;
```

Case Statement in SQL

CASE Statement can be utilised to restrictively go into some rationale in view of the situation with a condition being fulfilled. There are two sorts of CASE Statement:

Straightforward Case Explanation: used to go into some rationale in view of a strict worth.

Looked through Case Explanation: used to go into some rationale in view of the worth of an expression. The WHEN provision of the CASE proclamation characterises the worth that when fulfilled decides the progression of control.

Here is an illustration of a SQL methodology with a CASE explanation with a basic case-proclamation when-condition:

```
  CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
  LANGUAGE SQL
  BEGIN
DECLARE v_workdept CHAR(7);
SET v_workdepart = p_workdepart;
CASE  v_workdepart
      WHEN 'A1' THEN
        UPDATE department SET d_name = 'D1';
      WHEN 'B1' THEN
        UPDATE department SET d_name = 'D2';
      ELSE
        UPDATE department SET d_name = 'D3';
      END CASE
END
```

Here is an illustration of CASE proclamation with a looked case-explanation when-provision:

```
  CREATE PROCEDURE UPDATE_DEPART (IN p_workdepart)
  LANGUAGE SQL
  BEGIN
DECLARE v_workdept CHAR(3);
SET v_workdept = p_workdept;
CASE
WHEN v_workdept = 'A1' THEN
UPDATE department SET departname = 'D1';
WHEN v_workdept = 'B1' THEN
UPDATE department SET departname = 'D2';
ELSE
UPDATE department SET departname = 'D3';
END CASE
END
```

IF Proclamation in SQL

In the event that assertions can be utilised to restrictively go into some rationale in view of the situation with a condition being fulfilled. The IF proclamation is consistently identical to CASE explanations with a looked case-proclamation when proviso. The IF proclamation upholds the utilisation of discretionary ELSE IF provisos and a default ELSE condition. An END IF condition is expected to show the finish of the assertion.

Here is an illustration of procedure that contains an IF statement:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(10),
INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
IF rating = 2 THEN
UPDATE employee
SET salary = salary * 2.10, bonus = 2000
WHERE empno = empNum;
ELSEIF rating = 4 THEN
UPDATE employee
SET salary = salary * 2.05, bonus = 1000
WHERE empno = empNum;
ELSE
UPDATE employee
```

```
SET salary = salary * 1.03, bonus = 500
WHERE empno = empNum;
END IF;
END
```

CHARACTER FUNCTIONS

Character capabilities acknowledge character inputs and can return either characters or number qualities as a result. SQL gives various different person datatypes which incorporates – CHAR, VARCHAR,VARCHAR2, LONG, RAW, and LONG RAW. The different data types are arranged into three distinct data types:

- **VARCHAR2:** A variable-length character datatype whose information is changed over by the RDBMS.
- **Burn:** The fixed-length datatype.
- **Crude:** A variable-length datatype whose information isn't changed over by the RDBMS, yet entirely left in 'crude' structure.

At the point when a person's capability returns a person's esteem that worth is dependably of type VARCHAR2 (variable length), with the accompanying two exemptions: UPPER and LOWER. These capabilities convert to upper and to bring down case, individually, and return the CHAR values (fixed length), assuming that the strings they are approached to change over are **fixed-length** CHAR contentions.

SQL gives a rich arrangement of character works that permit you to get data about strings and change the items in those strings in more than one way.

- Character capabilities are of the accompanying two sorts:
- Case-Manipulative Functions (LOWER, UPPER and INITCAP)
- Character-Manipulative Functions (REPLACE, CONCAT, LENGTH, SUBSTR, INSTR, LPAD, RPAD, and TRIM)

Case-Manipulative Functions

- **Lower:** It has the ability switches alpha individual qualities over totally to lowercase. LOWER will really return a fixed-length string

in the event that the approaching string is fixed-length. LOWER won't change any characters in the string that are not letters, since the case is unimportant for numbers and unique characters, for example, the dollar sign ($) or modulus (%).

Syntax (Language Structure):

```
LOWER(SQL course)
Input1: SELECT LOWER('TechFORTech') FROM DUAL;
Output1: TechFORTech
Input2: SELECT LOWER('DATABASE@456') FROM DUAL;
Output2: database@456
```

- **Upper:** This capability switches alpha person values over completely to capitalised. Likewise, as well as UPPER capability will really return a fixed-length string on the off chance that the approaching string is fixed-length. UPPER won't change any characters in the string that are not letters, since the case is unessential for numbers and extraordinary characters, for example, the dollar sign ($) or modulus (%).

Syntax (Language Structure):

```
UPPER (SQL course)
Input1: SELECT UPPER ('techpoint') FROM DUAL;
Output1: techpoint
Input2: SELECT UPPER('Abcd $407%10') FROM DUAL;
Output2: Abcd $407%10
```

- **Initcap:** It has the capability to change alpha character values over completely to uppercase for the first letter of each word and rest others in lowercase. The words in the string should be isolated by either # or _ space. The words in the string should be isolated by either # or _ space.

Syntax (Sentence Structure):

```
INITCAP(SQL course)
Input1: SELECT INITCAP('Tech point is a software
engineering gateway for nerds') FROM DUAL;
```

```
Output1: Tech point is A Computer Science Portal For
Geeks
Input2: SELECT INITCAP ('LEARN_CODES_FOR_GOOD') FROM
DUAL;
Output2: Learn_Codes_For_Good
```

Character-Manipulative Functions

- **Concat:** This capability generally annexes (links) string2 to the furthest limit of string1. On the off chance that both of the string is NULL, CONCAT capability returns the non-NULL contention. On the off chance that the two strings are NULL, CONCAT brings NULL back.
- **Syntax:**

```
CONCAT('String1', 'String2')
Input1: SELECT CONCAT ('Data' ,'Learning') FROM DUAL;
Output1: DataLearning
Input2: SELECT CONCAT( NULL ,'Operating System') FROM
DUAL;
Output2: Operating System
Input3: SELECT CONCAT( NULL ,NULL ) FROM DUAL;
Output3: -
```

- **Length:** This capability returns the length of the information string. On the off chance that the info string is NULL, LENGTH capability returns NULL and not Zero. Likewise, assuming the information string contains additional areas towards the beginning, in the middle between or toward the finish of the string, then, at that point, the LENGTH capability incorporates the additional areas as well and returns the total length of the string.

Syntax:

```
LENGTH (Column|Expression)
Input1:  SELECT LENGTH ('Dancing Is Love ') FROM DUAL;
Output1: 20
Input2: SELECT LENGTH ('  Write an Experience ') FROM
DUAL;
Output2: 24
```

```
 Input3: SELECT LENGTH (‘’) FROM DUAL; or SELECT
LENGTH ( NULL ) FROM DUAL;
```

- **Substr:** It has capability returns a piece of a string from a given beginning point to an endpoint. In the event that a substring length isn't given, then SUBSTR returns every one of the characters till the finish of string (from the beginning position determined).

Syntax:

```
SUBSTR(‘String’,start-index,length_of_extracted_
string)
Input1: SELECT SUBSTR (‘Data management, 15) FROM
DUAL;
Output1: Data
Input2: SELECT SUBSTR (‘Data Manage System’, 9, 7)
FROM DUAL;
Output2:  Manage
```

- **Instr:** This capability returns numeric place of a person or a string in a given string. Alternatively, you can give a position m to begin looking, and the event n of string. Likewise, on the off chance that the beginning position isn't given, then, at that point, it begins search from record 1, as a matter of course. In the event that subsequent to looking through in the string, no match is found, INSTR capability brings 0 back.

Syntax:

```
INSTR (Column|Expression, ‘String’, [,m], [n])
Input: SELECT INSTR (‘Google applications are great
applications’,’app’,1,2) FROM DUAL;
Output: 23
```

- **LPAD and RPAD:** These capabilities return the strings cushioned to the left or right (according to the utilisation), consequently, the 'L' in 'LPAD' and the 'R' in 'RPAD', to a predefined length, and with a predetermined cushion string. On the off chance that the cushion string isn't indicated, then, at that point, the given string is cushioned on the left or right (according to the utilisation) with spaces.

Syntax:

```
LPAD(Column|Expression, n, 'String')
Syntax: RPAD(Column|Expression, n, 'String')
LPAD Input1: SELECT LPAD ('105',5,'*') FROM DUAL;
LPAD Output1: **105
LPAD Input2: SELECT LPAD('hey', 29, 'Tech') FROM DUAL;
LPAD Output2: TechTechTechTechhey
   RPAD Input1: SELECT RPAD('8000',7,'*') FROM DUAL;
RPAD Output1: 8000***
RPAD Input1: SELECT RPAD('Lern', 25, 'time') FROM DUAL;
RPAD Output1: Lerntimetimetimetime
```

- **Trim:** This capability manages the string input all along or end (or both). Assuming that no string or burn is indicated to be managed from the string and there exists some additional room at start or end of the string, then those additional areas are managed off

Syntax:

```
TRIM(Leading|Trailing|Both, trim_character FROM
trim_source)
Input1: SELECT TRIM('T' FROM 'TECH') FROM DUAL;
Output1: ECH
Input2: SELECT TRIM('techpoint') FROM DUAL;
Output2: Techpoint
```

- **Replace:** This capability looks for a person string and, whenever found, replaces it with a given substitution string at every one of the events of the string. Substitute is valuable for looking through examples of characters and afterwards changing all cases of that example in a solitary capability call. If a substitution string isn't given, then REPLACE capability eliminates every one of the events of that character string in the info string. In the event that neither a match string nor a substitution string is determined, REPLACE brings NULL

Syntax:

```
REPLACE(Text, search_string, replacement_string)
Input1: SELECT REPLACE('DATA MANAGEMENT',
'DATA','DATABASE') FROM DUAL;
```

```
Output1: DATABASE MANAGEMENT
Input2: SELECT REPLACE('abcdeabcccabdddeeabcc', 'ABC')
FROM DUAL;
Output2: deccabdddeec
```

Listing Function

The listing capability changes values from a gathering of columns into a rundown of values that are delimited by a configurable separator. Listing is regularly used to denormalise columns into a line of comma-isolated values (CSV) or other similar configurations reasonable for human perusing. Listing doesn't make a difference any getting away: it isn't by and large conceivable to tell whether an event of the separator in the outcome is a real separator, or simply an aspect of a worth. The protected utilisation of listing for electronic information connection points is hence restricted to cases in which an unambiguous separator can be chosen, for example, while amassing numbers, dates, or strings that are known to not contain the separator.

Syntax:

Listing is an arranged set capability, which requires the inside bunch statement to indicate a request. The insignificant language structure is as follows:

```
LISTAGG(<expression>, <separator>) WITHIN GROUP(ORDER
BY … )
```

The <expression> should not contain window capabilities, total capabilities, or subqueries.0 The standard just permits character literals in <separator> – for example, no articulation and no tight spot parameter.1 Bind boundaries are by and by all around upheld in practice. Listing eliminates invalid qualities before aggregation2 like most other total capabilities. In the event that no not invalid worth remaining parts, the aftereffect of listing is invalid. If necessary, blend can be utilised to supplant invalid qualities before conglomeration.

THE ON OVERFLOW STATEMENT

The return sort of listing is either varchar or clob with an execution characterised length limit. In practice, it is a varchar type. Listing acknowledges

the discretionary on flood provision to characterise the way of behaving assuming that the outcome surpasses the length furthest reaches of the bring type back:

```
LISTING(, ON OVERFLOW … )
```

The default is on flood blunder. For this situation, the standard requires an exemption with SQLSTATE 22001 to be brought – up practically speaking, this necessity isn't satisfied. The on flood shorten statement forestalls the flood by just linking; however, many qualities as the outcome type can oblige. Moreover, the on flood shorten condition permits one to determine how the outcome is ended:

```
ON OVERFLOW TRUNCATE [] WITH [OUT] COUNT
```

The discretionary defaults to three periods (...) and will be added as the last component in the event that truncation occurs. If with count is determined and truncation occurs, the quantity of overlooked values is placed in sections and annexed to the outcome. The SQL standard doesn't need an admonition to be given on truncation. To know regardless of whether the outcome is finished, clients can parse the result or look at the genuine length of the outcome to the determined length for an outcome containing all qualities.

Distinct

The listing capability acknowledges the discretionary set quantifiers all and particular:

```
LISTAGG( [ALL|DISTINCT] , … ) …
```

Assuming nor is determined, everything is default. Assuming particular is indicated, copy values are eliminated before conglomeration. Note that the end of copies is dependent upon the resemblance basically. Particular can be carried out physically by eliminating copies before collection – for example in a subquery. This works for data sets not supporting unmistakable in listing and furthermore permits to keep a specific event on the off chance that copies exist.

The accompanying model shows this methodology. The sections g and o address the gathering by and request by keys separately. The model

purposes min (o) to keep the main event on the off chance that one worth seems on various occasions.

```
SELECT g
     , LISTAGG(val, ‘,’) WITHIN GROUP (ORDER BY o) list
  FROM (SELECT g, min(o) o, Val
          FROM dist_listagg
         GROUP BY g, Val
       ) dt
 GROUP BY g
```

COMBINING LISTING WITH FILTER AND OVER

Listing can be combined with the filter and over clauses:

```
LISTING(…) WITHIN GROUP(…) [FILTER(WHERE …)] [OVER(…)]
```

The effect of the filter clause is to remove rows before aggregation. Case can be used for the same effect. The over clause must not contain an order by clause10 because the mandatory within group clause must contain an order by clause anyway. It is not possible to narrow the window frame: the set of aggregated rows is always the full partition.

COMPATIBILITY

Listing was introduced with SQL: 2016 as optional feature T625. Even though listing is not yet widely supported, most databases offer similar functionality using a proprietary syntax.

ARRAYS

In the event that the question doesn’t rigorously need the arrival of a delimited string, exhibits can be utilised to return a variety of values. A cluster can be developed utilising the array_agg total capability or by means of a subquery.

```
ARRAY_AGG( ORDER BY … )
ARRAY()
```

The subsequent structure can contain particular and bring first to eliminate copies and breaking point the exhibit length. Neither one nor the other methodologies play out a verifiable cast: The exhibit components

have a similar kind as. That implies that the recovering application can get the qualities in a sort of safe way and apply designing whenever required. The sort safe nature of exhibits permits them to likewise convey invalid qualities in an unambiguous manner. Array_agg does subsequently not eliminate invalid qualities like other total capabilities do (counting listing). The channel provision can be utilised to eliminate invalid qualities before conglomeration with array_agg. On the off chance that the channel condition eliminates all columns, array_agg returns invalid – not an unfilled exhibit. The subquery sentence structure permits eliminating invalid qualities in the where proviso of the and returns a vacant exhibit if the subquery returns no lines. On the off chance that the request for components is superfluous, multisets and gather can likewise be utilised to pass a sort of safe rundown to an application.

DOCUMENT TYPES

Like array_agg, the SQL standard characterises total capabilities that return JSON or XML pieces: for example, json_arrayagg and xmlagg. The principal benefit contrasted with listing is that they apply the separate getaway rules.

```
JSON_ARRAYAGG( ORDER BY … [NULL ON NULL])
XMLAGG(XMLELEMENT(NAME , ) ORDER BY …)
```

Using with Recursive

Albeit the listing usefulness can be carried out utilising with recursive, it is much of the time the better decision to utilise exhibits, reports or the restrictive options to listagg as displayed below. The following unique case can be executed utilising just with recursive and middle of the road SQL-92:

```
LISTING (DISTINCT ,  …) WITHIN GROUP(ORDER BY )
```

The example below makes use of g as group by key, value as

```
, and ‘,’as
DUE TO RECURSIVE
list_agg(g, val, list)
AS (
 SELECT g, min(val), CAST(null AS VARCHAR(255))
FROM listagg_demo
```

```
GROUP BY g
UNION ALL
SELECT prev.g,
(SELECT min(val)
FROM listagg_demo this
 WHERE this.g   = prev.g
AND this.Val > prev.val) Val
, COALESCE(list || ', ' || val
FROM list_agg prev
WHERE prev.Val IS NOT NULL
)

SELECT g, list
FROM list_agg
WHERE val IS NULL
ORDER BY g
```

This specific execution utilises the 'free record filter' procedure and the presentation will stay at a fairly low level even with a file on (g, val). The particular way of behaving is a symptom of this technique. The right treatment of invalid in Val is a significant exceptional case: albeit invalid is by and large disregarded in totals, a gathering that comprises of invalid qualities just should in any case be available in the outcome. This implies that invalid should not be taken out in the event that there is no not invalid worth in the gathering. The execution above utilises min(Val) in the non-recursive articulation to get this social more conventional execution that upholds all semantics and erratic request by conditions is conceivable utilising with recursive and window capabilities. Aaron Bertrand's post 'Assembled Concatenation in SQL Server' presents an illustration of this methodology.[1] In the two cases, erratic on flood conduct can be executed.

Proprietary Extensions

The main helpful expansion that is generally accessible is the help of tie boundaries and consistent articulations in. The standard neither permits precluding the nor discarding the inside bunch provision. However, a few information bases treat them as discretionary and apply execution

characterised defaults or uncover indistinct way of behaving if inside bunch is precluded.

Proprietary Alternatives

There are two broadly accessible restrictive options to listagg: group_concat and string_agg. Despite the fact that a few data sets utilise a similar exclusive capability name, they actually utilise an alternate sentence structure. Fortunately, the exclusive capabilities have a similar default semantic as listing: they channel invalid qualities before collection yet don't eliminate copies.

AGGREGATE FUNCTION IN SQL

Aggregate functions are unquestionably one of the many interesting features – well, functions – of SQL. Although they are not unique to SQL, they are frequently used. They are a component of the SELECT statement, which combines the strength of these operations with the advantages of SELECT (joining tables, filtering only the data and columns we require). A total capability in SQL plays out a computation on various qualities and returns a solitary worth. SQL gives many total capabilities that incorporate avg, count, aggregate, min, max, and so forth. A total capability overlooks NULL qualities when it plays out the computation, aside from the count capability. A total capability in SQL returns one worth subsequent to working out different upsides of a section. We frequently utilise total capabilities with the GROUP BY and HAVING conditions of the SELECT

Different sorts of SQL total capabilities are as follows:

- Count()
- Sum()
- Avg()
- Min()
- Max()

The accompanying table shows the SQL Server total capabilities:

Aggregate Function	Description
AVG	The AVG () total capability computes the normal of non-NULL qualities in a set.
CHECKSUM_AGG	The CHECKSUM_AGG () capability computes a checksum esteem in a group of rows.
COUNT	The COUNT () total capability returns the quantity of columns in a gathering, incorporating lines with NULL qualities.
COUNT_BIG	The COUNT_BIG () total capability returns the number of rows (with BIGINT information type) in a gathering, incorporating columns with NULL qualities.
MAX	The MAX () total capability returns the highest value (greatest) in a bunch of non-NULL qualities.
MIN	The MIN () total capability returns the lowest value (least) in a bunch of non-NULL qualities.
STDEV	The STDEV () capability returns the measurable standard deviation of all values given in the expression based on a sample of the data population.
STEVE	The STDEVP() capability likewise returns the standard deviation for all qualities in the given expression, however does so in light of the whole information populace.
SUM ()	The SUM () total capability returns the summation of all non-NULL qualities a set.
VAR	The VAR () capability returns the factual fluctuation of values in an articulation in light of an example of the predefined populace.
CARP	The VARP () capability returns the measurable difference of values in an articulation however does so in view of the whole information populace.

DATE FUNCTIONS

One of the most crucial SQL functions is the date, but it can be challenging for beginners to understand because there are numerous forms in which dates can be stored in databases and numerous formats in which users want to access dates depending on the particular needs. When storing both the date and time values at once in a single column in SQL, Date Time (time is also used together with the date) is usually utilised. Date time is sometimes used in place of dates because dates and times are related.

Let's understand each date function used in SQL one by one in detail:

Function	Description with Examples
NOW()	Returns the current date and time Query: Select NOW(); Returns: 2022-06-22 11:17:52
CURATE()	Returns the current date Query: Select CURDATE(); Returns: 2022-06-22
OUR TIME()	Returns the current time Query: Select CURTIME(); Returns: 11:17:52
DATE()	Separates the date part of a date or date/time articulation Info Table: Id | Name | Birth Time 120 | Patrick | 1998-09-16 10:40:15 Query: SELECT Name, DATE(Birth Time) AS Birthdate FROM Info; Output: Name | Birthdate Patrick | 1998-09-16
EXTRACT()	Returns a solitary piece of a date/time Syntax: EXTRACT(unit FROM date); Only some of the possible units – including the microsecond, second, minute, hour, day, week, month, quarter, and year – are actually utilised. Also an acceptable date phrase is "date. For instance, consider the "Info" table below: Id | Name | Birth Time 120 | Patrick | 1998-09-16 10:40:15 Query 1: SELECT Name, Extract(DAY FROM BirthTime) AS BirthDay FROM Info; Output: Name | Birthday Patrick | 16

(*Continued*)

Function	Description with Examples
	Query 2: SELECT Name, Extract(YEAR FROM BirthTime) AS BirthYear FROM Info; Output: Name: Patrick; Birth year: 1998 Query 3: SELECT Name, Extract(SECOND FROM BirthTime) AS BirthSecond FROM Info; Output: Name: Patrick; Birth year: 15
DATE_ADD()	Adds a predefined time stretch to a date Syntax: DATE_ADD(date, INTERVAL expr type); Where expr is the number of intervals to be added and date is a valid date expression and the following categories, and types are acceptable: Minute, Second, Second, Hour, Day, Week, Month, Quarter, Year, etc. For instance, consider the 'Info' table below. Id: 120; Name: Patrick; Birth Time: 1998-09-16 10:40:15 Query 1: SELECT Name, DATE_ADD(BirthTime, INTERVAL 1 YEAR) AS BirthTimeModified FROM Test; Output: Id: 120; Name: Patrick; Birth Time: 1999-09-16 Query 2: SELECT Name, DATE_ADD(BirthTime, INTERVAL 30 DAY) AS BirthDayModified FROM Test; Output: Id: 120; Name: Patrick; Birth Time: 1999-10-16
DATE_SUB()	Deducts a predetermined time stretch from a date Query: Select DATE_SUB();
DATEDIFF()	Returns the quantity of days between two dates Syntax: DATEDIFF(date1, date2); date1 & date2- date/time expression Query: SELECT DATEDIFF('2018-01-10','2019-01-20') AS DateDiff; Output: 10
DATE_ FORMAT()	Displays date/time data in different formats Syntax: DATE_FORMAT(date,format); Query: DATE_FORMAT(NOW(),'%d %b %y'; Output:10 Jan 2017

Date is a substantial date and configuration indicates the result design for the date/time. The configurations that can be utilised are:

%a-Abbreviated work day name (Sun–Sat)

%b-Abbreviated month name (Jan–Dec)

%c-Month, numeric (0–12)

%D-Day of month with English postfix (0th, first, second, third)

%d-Day of month, numeric (00–31)

%e-Day of month, numeric (0–31)

%f-Microseconds (000000–999999)

%H-Hour (00–23)

%h-Hour (01–12)

%I-Hour (01–12)

%I-Minutes, numeric (00–59)

%j-Day of year (001–366)

%k-Hour (0–23)

%l-Hour (1–12)

%M-Month name (January–December)

%m-Month, numeric (00–12)

%p-AM or PM

%r-Time, 12-hour (hh:mm:ss followed by AM or PM)

%S-Seconds (00–59)

%s-Seconds (00–59)

%T-Time, 24-hour (hh:mm:ss)

%U-Week (00–53) where Sunday is the primary day of week

%u-Week (00–53) where Monday is the primary day of week

%V-Week (01–53) where Sunday is the primary day of week, utilised with %X

%v-Week (01–53) where Monday is the main day of week, utilised with %x

%W-Weekday name (Sunday–Saturday)

%w-Day of the week (0=Sunday, 6=Saturday)

%X-Year for the week where Sunday is the principal day of week, four digits, utilised with %V

%x-Year for the week where Monday is the primary day of week, four digits, utilised with %v

%Y-Year, numeric, four digits

%y-Year, numeric, two digits

Some of the fundamental date functions used in SQL are those that were just mentioned. Other date functions are utilised in various contexts. Before utilising any of them, one must be aware of the syntax and arguments supplied into the function to prevent unexpected outcomes.

NULL VALUES IN SQL

A field with a NULL value has no value at all. It is possible to create a new record or update an existing record without providing a value for a field that is optional in a table. The field will then be saved with the value NULL. A column is said to be null, or to contain null, if it contains no value in a row of columns. Sections of any information type that are not obliged by NOT NULL or PRIMARY KEY uprightness imperatives can contain nulls. When the true value is unknown or when a value would be meaningless, use a null. Since NULL is not itself a value and lacks a data type, it serves as a placeholder for attributes that have uncertain or missing values. Codd suggested separating NULLs into two categories:

- A-marks: Info that could be relevant but is unknown (such as someone's age).
- I-marks indicate that the data are not appropriate (for example, a phone number for someone without a phone or a spouse's name for someone not married).

Oracle Database considers a character value with a length of zero to be invalid. However, as they are not equal, they do not use null to represent a numeric value of zero. Null is the result of any arithmetic expression that contains one. For instance, null is null when added to 10. In actuality, when given a null operand, all operators (apart from concatenation) return null.

Why Are Null Functions Necessary?

To perform operations on the null values kept in our database, null functions are necessary. On NULL values, we can run functions that explicitly identify if a value is null or not.

With the use of this ability to recognise null values, additional actions on them can be carried out, much like the aggregate procedures in SQL. The following are a few of the functions:

Sr. No.	Function	Description
1	ISNULL()	Assists us to supplant NULL qualities with the desired worth.
2	IFNULL()	Permits us to return the main worth on the off chance that the worth is NULL, and in any case returns the subsequent worth.
3	COALESCE()	Assists us with returning the first non-invalid qualities in quite a while.
4	NVL()	Assists with supplanting the NULL worth with the ideal worth given by the client.

How Can Null Values Be Tested?

With comparison operators like =, <, or<.it is not possible to check for NULL values. We will have to use the IS NULL and IS NOT NULL operators instead.

Is Null Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

Is Not Null Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

Syntax:
The basic syntax of NULL while creating a table:

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

NOT NULL in this case denotes that the column must always accept an explicit value of the specified data type. We did not apply NOT NULL in two columns; hence, these columns are open to NULL values. When a field contains a NULL value, it means that the record creation process left the field empty.

However, when comparing an unknown number to any other value, the result is always unknown and not included in the final results, which is why the NULL value might be problematic when picking data. To check for a NULL value, you must use the IS NULL or IS NOT NULL operators. Take a look at the CUSTOMERS table, which contains the following records:

ID	Name	Age	Address	Salary
1022	Ravish	32	Los Angles	7,000.00
1028	Kevin	23	Texas	8,000.00
1030	Chris	25	Ohio	6,500.00
1042	Humpty	27	Omaha	5,500.00
1051	Kevin	22	Manchester	
1055	Khalessi	25	California	3,500.00
1060	Maven	24	San Francisco	

The IS NOT NULL operator is now used as follows:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
WHERE SALARY IS NOT NULL;
```

Output:

ID	Name	Age	Address	Salary
1022	Ravish	32	Los Angles	7,000.00
1028	Kevin	23	Texas	8,000.00
1030	Chris	25	Ohio	6,500.00
1042	Humpty	27	Omaha	5,500.00
1055	Khalessi	25	California	3,500.00

The IS NULL operator is now used as follows:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
WHERE SALARY IS NULL;
```

Output:

ID	Name	Age	Address	Salary
1051	Kevin	22	Manchester	
1060	Maven	24	San Francisco	

NUMERIC FUNCTIONS

The numerical functions accept a numeric expression as input and output numeric results. The majority of mathematical functions have NUMBER as their return type. The following table lists the included numerical functions:

Function	Description
ABS(***numeric_exp*** **)**	Returns the outright worth of *numeric_exp*.
ACOS(***float_exp*** **)**	Returns the arccosine of *float_exp* as a point, communicated in radians.
ASIN(***float_exp*** **)**	Returns the arcsine of *float_exp* as a point, communicated in radians.
ATAN(***float_exp*** **)**	Returns the arctangent of *float_exp* as a point, communicated in radians.
ATAN2(***float_exp1, float_exp2*** **)**	Returns the arctangent of the *x* and, not set in stone by *float_exp1* and *float_exp2*, independently, as a point, imparted in radians.

(*Continued*)

Function	Description
CEILING(*numeric_exp*)	Returns the littlest number more noteworthy than or equivalent to *numeric_exp*. The return esteem is of similar information type as the info boundary.
COS(*float_exp*)	gives the cosine of *float exp*, where *float* exp is the radian-based coordinate of the point.
COT(*float_exp*)	Returns the cotangent of *float_exp*, where *float_exp* is a point communicated in radians.
DEGREES(*numeric_exp*)	Returns the quantity of degrees changed over from *numeric_exp* radians.
EXP(*float_exp*)	Returns the exponential value of *float_exp*.
FLOOR(*numeric_exp*)	Returns the biggest number not exactly or equivalent to *numeric_exp*. The return esteem is of similar information type as the information parameter.
LOG(*float_exp*)	returns the float exp's natural logarithm.
LOG10(*float_exp*)	gives back the base The float exp 10 logarithm.
MOD(*integer_exp1, integer_exp2*)	Returns the amount left over (modulus) after dividing *integer exp1* by *integer exp2*.
PI( )	gives a floating-point value that is equal to the constant value of pi.
POWER(*numeric_exp, integer_exp*)	returns the result of *scaling numeric* exp by *integer exp*.
RADIANS(*numeric_exp*)	gives the number of radians created when *numeric exp* degrees were converted.
RAND([*integer_exp*])	Returns a random floating-point number using the specified seed value of integer exp.
ROUND(*numeric_exp, integer_exp*)	returns *the numeric exp* rounded to *integer exp* places to the right of the decimal. *Numeric exp* is rounded to |*integer exp*| places to the left of the decimal point if integer exp is negative.
SIGN(*numeric_exp*)	gives back a numeric exp sign signal. –1 is returned if *numeric exp* is less than zero. 0 is returned if *numeric exp* is equal to zero. One is returned if *numeric* exp is larger than 0.
SIN(*float_exp*)	Returns the sine of *float_exp*, where *float_exp* is an angle expressed in radians.
SQRT(*float_exp*)	Returns the square root of *float_exp*.
TAN(*float_exp*)	Returns the tangent of *float_exp*, where *float_exp* is an angle expressed in radians.
TRUNCATE(*numeric_exp, integer_exp*)	Returns *numeric_exp* shortened to *integer_exp* puts right of the decimal point. Assuming *integer_exp* is negative, *numeric_exp* is shortened to |*integer_exp*| spots to one side of the decimal point.

With the exception of ABS, ROUND, TRUNCATE, SIGN, FLOOR, and CEILING, which return upsides of similar information type as the information boundaries, all numerical capabilities return upsides of the sort SQL FLOAT. The value of another scalar function, the name of a column, or a numeric literal can all be used as arguments for the numeric exp function. The underlying data type can be one of the following: SQL NUMERIC, SQL TINYINT, SQL SMALLINT, SQL INTEGER, SQL BIGINT, SQL FLOAT, SQL DECIMAL, SQL REAL, or SQL DOUBLE. The name of a column, the output of another scalar function, or a numeric literal with SQL FLOAT as the underlying data type can all be used as float exp arguments.

STRING FUNCTIONS

A string function always accepts a string value as an input, no matter what data type is returned. There are numerous built-in string functions in SQL Server that developers can use. The SQL Server string functions listed below examine an input string and output a string or a number:

Function	Description
ASCII	Return the value of the character in terms of ASCII code value
CHAR	Convert the value of the character as an ASCII value
CHARINDEX	Look for a substring inside a string beginning from a predefined area and return the place of the substring
CONCAT	join at least two strings into one string
CONCAT_WS	Connect various strings with a separator into a solitary string
DIFFERENCE	Analyse the SOUNDEX() upsides of two strings
FORMAT	Return a worth designed with the predefined design and discretionary culture
LEFT	Separate a given various characters from a character string beginning from the left
LEN	Return various characters of a character string
LOWER	Lowercase a string by changing it
LEITRIM	Remove all leading blanks from the provided string to produce a new string.
NEAR	Return the Unicode character that corresponds to the requested integer code in accordance with the Unicode standard.
PATINDEX	The starting location of a pattern's first appearance in a string is returned.

(*Continued*)

Function	Description
QUOTE NAME	provides a response that is a Unicode string with the delimiters appended to make the input string a legitimate delimited identifier.
REPLACE	Anywhere a substring appears in a string, replace it with a different substring.
REPLICATE	Get a string that has been repeated a certain amount of times.
REVERSE	Return a character string's reverse order.
RIGHT	Extract a specified number of characters, starting on the right, from a character string.
RTRIM	Remove any trailing blanks from the provided string to create a new string.
SOUNDEX	If a string is spoken, return a four-character (SOUNDEX) code for it.
SPACE	Returns a string of repeated spaces.
STR	Returns character data converted from numeric data.
STRING_AGG	Concatenate rows of strings with a specified separator into a new string
STRING_ESCAPE	Escapes special characters in a string and returns a new string with escaped characters
STRING_SPLIT	A table-valued function that splits a string into rows of substrings based on a specified separator.
STUFF	Delete a part of a string and then insert another substring into the string starting at a specified position.
SUBSTRING	Extract a substring within a string starting from a specified location with a specified length
TRANSLATE	Replace several single-characters, one-to-one translation in one operation.
TRIM	To make a new string, take off the leading and trailing blanks from the given string.
UNICODE	returns a character's integer value according to the Unicode standard.
UPPER	Make a string all uppercase.

Except for FORMAT, all built-in string functions are deterministic. This indicates that they always produce the same result when called with a particular set of input values.

The input type is implicitly changed to a text data type when inputs to string functions are not strings.

Deterministic and Nondeterministic

When invoked with a certain set of input values and the same database state, deterministic functions always provide the same output. Even though the database state that they access remains constant, nondeterministic functions may produce different outcomes each time they are called with a particular set of input values. Given the aforementioned restrictions, the function AVG, for instance, consistently produces the same outcome, yet the GETDATE function, which returns the current date time value, consistently produces a different outcome.

In order for the SQL Server Database Engine to be able to index the results of user-defined functions, either through indexes on computed columns that call the function or through indexed views that reference the function, a number of criteria must be met. One of these characteristics is a function's determinism. For instance, if a view contains any references to nondeterministic functions, a clustered index cannot be generated on that view.

Built-in Function Determinism

Any built-in function's determinism cannot be changed. Depending on how SQL Server implements the function, each built-in function is either deterministic or nondeterministic. For instance, adding an ORDER BY clause to a query doesn't alter the query's use of a deterministic function. With the exception of FORMAT, every built-in function for strings is deterministic. See String Functions for a list of these functions.

The following built-in functions are always predictable since they fall within built-in function categories other than string functions:

ABS, POWER, ACOS, DAY, RADIANS, ASIN, DEGREES, ROUND, ATAN, EXP, SIGN, ATN2, FLOOR, SIN, CEILING, ISNUMERIC, SQRT, LOG, TAN, COT, LOG10, YEAR, DATALENGTH, MONTH, NULLIF

The following functions can be used in indexed views or indexes on calculated columns even if they are not always deterministic when they are provided that way.

Function	Comments
All aggregate	These are deterministic unless they are specified with the OVER and ORDER BY clauses.
CAST	Deterministic unless used with **sql_variant, small date time**, or **DateTime**.
CONVERT	Deterministic unless any of the following circumstances apply: **SQL variant** is the source type. The source type is nondeterministic, and the target type is **SQL variant**. **Datetime** or **smalldatetime** is the source or target type, a character string is the other source or target type, and a nondeterministic style is given. The style parameter must be a constant for the system to be deterministic. Except for styles 20 and 21, all styles with a value of less than or equal to 100 are nondeterministic.
	With the exception of styles 106, 107, 109, and 113, all styles larger than 100 are deterministic.
	With the exception of CHECKSUM(*), CHECKSUM is deterministic.
	Only when used in conjunction with the CONVERT function, ISDATE is deterministic for however long style isn't equivalent to 0, 100, 9, or 109 and the CONVERT style boundary is provided. Only when a *seed* parameter is supplied, RAND is deterministic.

The system's statistical functions, metadata, security, and configuration are all nondeterministic.

SUMMARY

In the chapter, we learned about the functions, which are pre-written programmes that may take variables and return a value, and Oracle offers various built-in functions. Function can be used alone or can be used in combination with several other functions like string, math, date, translation etc. There may be several uses for a function depending on the kind of information that is given to carry out code.

NOTE

1 Grouped Concatenation in SQL Server- Aaron Bertrand, SQL Performance.com

Bibliography

Adrienne Watt & Nelson Eng - SQL Structured query language - https://opentextbc.ca/dbdesign01/chapter/sql-structured-query-language/. Accessed on (2022 June 11).

Adrienne Watt & Nelson Eng - SQL Structured query language - https://opentextbc.ca/dbdesign01/chapter/sql-structured-query-language/. Accessed on (2022 June 16).

Ahmad Yaseen - Constraints in SQL Server: SQL NOT NULL, UNIQUE and SQL primary key - https://www.sqlshack.com/commonly-used-sql-server-constraints-not-null-unique-primary-key/. Accessed on (2022 June 17).

Ajay Sarangam - Important types of indexes in SQL server - https://u-next.com/blogs/data-science/types-of-indexes-in-sql-server/. Accessed on (2022 June 14).

Andrew Pomponio - MySQL overview: Key features, benefits, and use cases - https://www.openlogic.com/blog/mysql-overview. Accessed on (2022 June 16).

Chad Brooks - When to use SQL - https://www.businessnewsdaily.com/5804-what-is-sql.html. Accessed on (2022 June 10).

Data Flow - https://learn.microsoft.com/en-us/sql/integration-services/data-flow/data-flow?view=sql-server-ver16. Accessed on (2022 June 17).

How SQL works? - https://www.tutorialspoint.com/sql/sql-overview.htm. Accessed on (2022 June 12).

Index in SQL - https://www.simplilearn.com/tutorials/sql-tutorial/index-in-sql. Accessed on (2022 June 14).

Installing MySQL on Microsoft Windows - https://dev.mysql.com/doc/refman/8.0/en/windows-installation.html. Accessed on (2022 June 16).

Introduction to SQL - https://www.w3schools.com/sql/sql_intro.asp. Accessed on (2022 June 10).

Kashyap Vyas - Major Advantages of Using MySQL - https://www.datamation.com/storage/8-major-advantages-of-using-mysql/. Accessed on (2022 June 15).

Kate Brush - RDBMS (relational database management system). -https://www.techtarget.com/searchdatamanagement/definition/RDBMS-relational-database-management-system. Accessed on (2022 June 10).

Katie Terrell Hanna| Sarah Lewis - SQL injection - https://www.techtarget.com/searchsoftwarequality/definition/SQL-injection. Accessed on (2022 June 15).

MySQL advantages and disadvantages - https://www.techstrikers.com/MySQL/advantages-and-disadvantages-of-mysql.php. Accessed on (2022 June 16).
MySQL advantages and disadvantages - https://www.w3schools.blog/mysql-advantages-disadvantages. Accessed on (2022 June 16).
MySQL Enterprise Edition - https://www.mysql.com/downloads. Accessed on (2022 June 16).
MySQL Exercises - https://www.w3schools.com/MySQL/default.asp. Accessed on (2022 June 15).
MySQL Features - https://en.wikipedia.org/wiki/MySQL. Accessed on (2022 June 15).
MySQL Functions - https://www.techonthenet.com/mysql/functions.php#:~:text=In%20MySQL, %20a%20function%20is, and%20then%20return%20a%20value. Accessed on (2022 June 15).
MySQL HeatWave - https://www.mysql.com/. Accessed on (2022 June 15).
MySQL - Installation - https://www.tutorialspoint.com/mysql/mysql-installation.htm. Accessed on (2022 June 17).
MySQL RDBMS https://www.w3schools.com/mysql/mysql_rdbms.asp#:~:text=RDBMS%20stands%20for%20Relational%20Database, the%20data%20in%20the%20database. Accessed on (2022 June 13).
MySQL Tutorial - https://www.tutorialspoint.com/mysql/index.htm. Accessed on (2022 June 15).
Naveen - What is a database management system? - https://intellipaat.com/blog/tutorial/sql-tutorial/introduction-to-sql/. Accessed on (2022 June 11).
Peter Loshin - Structured query language (SQL). - https://www.techtarget.com/searchdatamanagement/definition/SQL. Accessed on (2022 June 10).
PHP MySQL INSERT query - https://www.tutorialrepublic.com/php-tutorial/php-mysql-insert-query.php. Accessed on (2022 June 18).
PHP MySQL use the WHERE clause https://www.w3schools.com/php/php_mysql_select_where.asp. Accessed on (2022 June 18).
PL/SQL - Environment setup - https://www.tutorialspoint.com/plsql/plsql_environment_setup.htm. Accessed on (2022 June 12).
Rajendra Gupta - An overview of SQL comments - https://www.sqlshack.com/an-overview-of-sql-comments/. Accessed on (2022 June 17).
Richard Peterson - What is SQL? Learn SQL basics, SQL full form & how to use - https://www.guru99.com/what-is-sql.html. Accessed on (2022 June 10).
Saurabh Hooda - What are the advantages of a database management system? - https://www.goskills.com/Development/Resources/Advantages-of-database-management-system. Accessed on (2022 June 11).
Richard Peterson - SQL commands: DML, DDL, DCL, TCL, DQL with query example - https://www.guru99.com/sql-commands-dbms-query.html. Accessed on (2022 June 12).
Shailendra Chauhan - Different types of SQL server functions https://www.dotnettricks.com/learn/sqlserver/different-types-of-sql-server-functions#:~:text=A%20function%20is%20a%20database,the%20database%20table(s).. Accessed on (2022 June 16).

Siddharth Sachdeva - Query processing - https://www.analyticsvidhya.com/blog/2021/10/a-detailed-guide-on-sql-query-optimization/. Accessed on (2022 June 13).

Siddharth Sachdeva - SQL query optimization - https://www.analyticsvidhya.com/blog/ 2021/10/a-detailed-guide-on-sql-query-optimization/. Accessed on (2022 June 13).

SQL Clauses and its types - Syntax and example - https://data-flair.training/blogs/sql-clauses/. Accessed on (2022 June 13).

SQL constraints - https://www.w3schools.com/sql/sql_constraints.asp. Accessed on (2022 June 13).

SQL Constraints - https://www.w3schools.com/sql/sql_constraints.asp#:~:text=SQL%20constraints%20are%20used%20to%20specify%20rules%20for%20the%20data, action, %20the%20action%20is%20aborted. Accessed on (2022 June 17).

SQL Create database statement - https://www.w3schools.com/sql/sql_create_db.asp. Accessed on (2022 June 13).

SQL Data types for MySQL, SQL Server, and MS access -https://www.w3schools.com/sql/sql_datatypes.asp. Accessed on (2022 June 14).

SQL - Expressions - https://www.tutorialspoint.com/sql/sql-expressions.htm. Accessed on (2022 June 13).

SQL injection - https://portswigger.net/web-security/sql-injection. Accessed on (2022 June 14).

SQL Injection attacks (SQLi). - https://www.rapid7.com/fundamentals/sql-injection-attacks/. Accessed on (2022 June 15).

SQL Injection based on batched SQL statements - https://www.w3schools.com/sql/sql_injection.asp. Accessed on (2022 June 16).

SQL - Operators - https://www.javatpoint.com/sql-operators. Accessed on (2022 June 13).

SQL - Operators - https://www.w3schools.com/sql/sql_operators.asp. Accessed on (2022 June 13).

SQL Server functions - https://www.javatpoint.com/sql-server-functions. Accessed on (2022 June 17).

SQL Server functions - https://www.w3schools.com/sql/sql_ref_sqlserver.asp. Accessed on (2022 June 17).

SQL Server hardening best practices - https://www.netwrix.com/sql_server_security_best_practices.html. Accessed on (2022 June 15).

SQL - Syntax - https://www.tutorialspoint.com/sql/sql-syntax.htm. Accessed on (2022 June 13).

SQL (Structured query language). injection - https://www.imperva.com/learn/application-security/sql-injection-sqli/. Accessed on (2022 June 16).

Syntax - https://en.wikipedia.org/wiki/SQL#:~:text=SQL%20was%20initially%20developed%20at, Codd%20in%20the%20early%201970s. Accessed on (2022 June 11).

Tehreem Naeem - Relational database management systems (RDBMS). - https://www.astera.com/type/blog/relational-database-management-system/. Accessed on (2022 June 11).

Timeline of MySQL - https://www.tutorialspoint.com/discuss-the-history-of-mysql. Accessed on (2022 June 15).
Types of SQL injections - https://www.imperva.com/learn/application-security/sql-injection-sqli/#:~:text=Types%20of%20SQL%20Injections, data%20and%20their%20damage%20potential. Accessed on (2022 June 14).
Type of SQL statements - https://way2tutorial.com/sql/type-of-sql-statements.php. Accessed on (2022 June 12).
Types of SQL statements - https://docs.oracle.com/cd/B14117_01/server.101/b10759/statements_1001.htm. Accessed on (2022 June 12).
Vijay Kanade - What is SQL? Definition, elements, examples - https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-sql/. Accessed on (2022 June 11).
What is an SQL injection attack? - https://www.rapid7.com/fundamentals/sql-injection-attacks/. Accessed on (2022 June 14).
What is Database? - https://www.javatpoint.com/mysql-tutorial. Accessed on (2022 June 15).
What is MySQL? - https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html. Accessed on (2022 June 16).
What Is MySQL? A beginner-friendly explanation - https://kinsta.com/knowledgebase/what-is-mysql/. Accessed on (2022 June 16).
What is SQL injection (SQLi). and How to Prevent It - https://www.acunetix.com/websitesecurity/sql-injection/. Accessed on (2022 June 16).
What is SQL Injection? - https://www.veracode.com/security/sql-injection. Accessed on (2022 June 15).
What is the impact of a successful SQL injection attack? - https://portswigger.net/web-security/sql-injection. Accessed on (2022 June 16).

Index

Note: *Italic* page numbers refer to figures.