**1.   Query for Retrieving Tables**

This query can be run to retrieve the list of tables present in a database where the database is “My\_Schema”.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM My\_Schema.Tables; |

**2.**[Query for Selecting Columns from a Table](https://bytescout.com/blog/deep-sql-queries-and-examples.html#2)

This is perhaps the most widely used of SQL queries examples. In the example below, we are extracting the “Student\_ID” column or attribute from the table “STUDENT”.

|  |  |
| --- | --- |
| 1 | SELECT Student\_ID FROM STUDENT; |

If you want to display all the attributes from a particular table, this is the right query to use:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM STUDENT; |

**3.**[Query for Outputting Data Using a Constraint](https://bytescout.com/blog/deep-sql-queries-and-examples.html#3)

This SQL query retrieves the specified attributes from the table on the constraint *Employee ID =0000*

|  |  |
| --- | --- |
| 1 | SELECT EMP\_ID, NAME FROM EMPLOYEE\_TBL WHERE EMP\_ID = '0000'; |

**4. Query for Outputting Sorted Data Using ‘Order By’**

This query orders the results with respect to the attribute which is referenced to using “Order By” – so for example, if that attribute is an integer data type, then the result would either be sorted in ascending or descending order; likewise, if the data type is a String then the result would be ordered in alphabetical order.

|  |  |
| --- | --- |
| 1  2 | SELECT EMP\_ID, LAST\_NAME FROM EMPLOYEE  WHERE CITY = 'Seattle' ORDER BY EMP\_ID; |

The ordering of the result can also be set manually, using “asc ” for ascending and “desc” for descending.

|  |  |
| --- | --- |
| 1  2 | SELECT EMP\_ID, LAST\_NAME FROM EMPLOYEE\_TBL  WHERE CITY = 'INDIANAPOLIS' ORDER BY EMP\_ID asc; |

**5.   Query for Outputting Sorted Data Using ‘Group By’**

The ‘Group By’ property groups the resulting data according to the specified attribute.

|  |  |
| --- | --- |
| 1  2 | SELECT Name, Age FROM Patients WHERE Age > 40  GROUP BY Age ORDER BY Name; |

**SQL Queries for Data Manipulation Using Math Functions**

There are a lot of built-in math functions like COUNT and AVG which provide basic functionalities of counting the number of results and averaging them respectively.

**6. Data Manipulation Using COUNT**

This query displays the total number of customers by counting each customer ID. In addition, it groups the results according to the country of each customer.

|  |  |
| --- | --- |
| 1 | SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country; |

**7.**[Data Manipulation Using SUM](https://bytescout.com/blog/deep-sql-queries-and-examples.html#7)

SUM calculates the total of the attribute that is given to it as an argument.

|  |  |
| --- | --- |
| 1 | SELECT SUM(Salary)FROM Employee WHERE Emp\_Age < 30; |

**8. Data Manipulation Using AVG**

Simple – an average of a given attribute.

|  |  |
| --- | --- |
| 1 | SELECT AVG(Price)FROM Products; |

**9.   Query for Listing all Views**

This SQL query lists all the views available in the schema.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM My\_Schema.views; |

**10. Query for Creating a View**

A view is a tailored table that is formed as a result of a query. It has tables and rows just like any other table. It’s usually a good idea to run queries in SQL as independent views because this allows them to be retrieved later to view the query results, rather than computing the same command every time for a particular set of results.

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE VIEW Failing\_Students AS  SELECT S\_NAME, Student\_ID  FROM STUDENT  WHERE GPA > 40; |

**11. Query for Retrieving a View**

The standard syntax of selecting attributes from a table is applicable to views as well.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Failing\_Students; |

**12. Query for Updating a View**

This query updates the view named ‘Product List’ – and if this view doesn’t exist, then the Product List view gets created as specified in this query.

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE OR REPLACE VIEW [ Product List] AS  SELECT ProductID, ProductName, Category  FROM Products  WHERE Discontinued = No; |

**13. Query for Dropping a View**

This query will drop or delete a view named ‘V1’.

|  |  |
| --- | --- |
| 1 | DROP VIEW V1; |

**14. Query to Display User Tables**

**A user-defined table** is a representation of defined information in a table, and they can be used as arguments for procedures or **user-defined functions**. Because they’re so useful, it’s useful to keep track of them using the following query.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.objects WHERE Type='u' |

**15. Query to Display Primary Keys**

A primary key uniquely identifies all values within a table. The following SQL query lists all the fields in a **table’s primary key.**

|  |  |
| --- | --- |
| 1 | SELECT \* from Sys.Objects WHERE Type='PK' |

**16. Query for Displaying Unique Keys**

**A Unique Key** allows a column to ensure that all of its values are different.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='uq' |

**17. Displaying Foreign Keys**

**Foreign keys link** one table to another – they are attributes in one table which refer to the primary key of another table.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='f' |

**18. Displaying Triggers**

A **Trigger** is sort of an ‘event listener’ – i.e, it’s a pre-specified set of instructions that execute when a certain event occurs. The **list of defined triggers** can be viewed using the following query.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='tr' |

**19. Displaying Internal Tables**

**Internal tables** are formed as a by-product of a **user-action** and are usually not accessible. The data in internal tables cannot be manipulated; however, the metadata of the internal tables can be viewed using the following query.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='it' |

**20. Displaying a List of Procedures**

A stored procedure is a **group of SQL queries** that logically form a single unit and perform a particular task. Thus, using the following query you can keep track of them:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='p' |

**.. and TWENTY More Advanced SQL Queries for our Users!**

**21. Swapping the Values of Two Columns in a table**

In this and subsequent examples, we will use a common company database including several tables which are easily visualized. Our practice DB will include a Customers table and an Order table. The Customers table will contain some obvious columns including ID, Name, Address, zip, and email, for example, where we assume for now that the primary key field for indexing is the *Customer\_ID* field.

With this in mind, we can easily imagine an Orders table which likewise contains the indexed customer ID field, along with details of each order placed by the customer. This table will include the order Number, Quantity, Date, Item, and Price. In our first one of **SQL examples**, imagine a situation where the zip and phone fields were transposed and all the phone numbers were erroneously entered into the zip code field. We can easily fix this problem with the following SQL statement:

|  |  |
| --- | --- |
| 1 | UPDATE Customers SET Zip=Phone, Phone=Zip |

**22.**[Returning a Column of Unique Values](https://bytescout.com/blog/deep-sql-queries-and-examples.html#22)

Now, suppose that our data entry operator added the same Customers to the Customers table more than once by mistake. As you know, proper indexing requires that the key field contain only unique values. To fix the problem, we will use *SELECT DISTINCT* to create an indexable list of unique customers:

|  |  |
| --- | --- |
| 1 | SELECT DISTINCT ID FROM Customers |

**23. Making a Top 25 with the SELECT TOP Clause**

Next, imagine that our Customers table has grown to include thousands of records, but we just want to show a sample of 25 of these records to demonstrate the column headings and The SELECT TOP clause allows us to specify the number of records to return, like a *Top-25 list*. In this example we will return the top 25 from our Customers table:

|  |  |
| --- | --- |
| 1 | SELECT TOP 25 FROM Customers WHERE Customer\_ID<>NULL; |

**24. Searching for SQL Tables with Wildcards**

Wildcard characters or operators like “%” make it easy to find particular strings in a large table of thousands of records. Suppose we want to find all of our customers who have names beginning with “Herb” including Herberts, and Herbertson. The **%** wildcard symbol can be used to achieve such a result. The following SQL query will return all rows from the Customer table where the **Customer\_name** field begins with “Herb”:

|  |  |
| --- | --- |
| 1 | SELECT \* From Customers WHERE Name LIKE 'Herb%' |

**25. Between Monday and Tuesday**

Today is Wednesday, and we arrive at work and discover that our new data entry clerk in training has entered all new orders incorrectly on Monday and Tuesday. We wish to teach our new trainee to find and correct all erroneous records. What’s the easiest way to get all the records from the Orders table entered on Monday and Tuesday? The Between clause makes the task a breeze:

|  |  |
| --- | --- |
| 1  2 | SELECT ID FROM Orders WHERE  Date BETWEEN ‘01/12/2018’ AND ‘01/13/2018’ |

**26.**[Finding the Intersection of Two Tables](https://bytescout.com/blog/deep-sql-queries-and-examples.html#26)

Undoubtedly the whole reason that a relational database exists in the first place is to find matching records in two tables! The JOIN statement accomplishes this core objective of SQL and makes the task easy. Here we are going to fetch a list of all records which have matches in the Customers and Orders tables:

|  |  |
| --- | --- |
| 1  2 | SELECT ID FROM Customers INNER  JOIN Orders ON Customers.ID = Orders.ID |

The point of INNER JOIN, in this case, is to select records in the Customers table which have a matching customer ID values in the Orders table and return only those records. Of course there are many types of JOIN, such as FULL, SELF, and LEFT, but for now, let’s keep things interesting and move on to more diverse types of queries.

**27. Doubling the Power with UNION**

We can combine the results of two SQL queries examples into one naturally with the UNION keyword. Suppose we want to create a new table by combining the Customer\_name and phone from Customers with a list of that customer’s recent orders so that we can look for patterns and perhaps suggest future purchases. Here is a quick way to accomplish the task:

|  |  |
| --- | --- |
| 1  2 | SELECT phone FROM Customers  UNION SELECT item FROM Orders |

The UNION keyword makes it possible to combine JOINS and other criteria to achieve very powerful new table generation potential.

**28.**[Making Column Labels More Friendly](https://bytescout.com/blog/deep-sql-queries-and-examples.html#28)

Aliasing column labels give us the convenience of renaming a column label to something more readable. There is a tradeoff when naming columns to make them succinct results in reduced readability in subsequent daily use. In our Orders table, the item column contains the description of purchased products. Let’s see how to alias the item column to temporarily rename it for greater user-friendliness:

|  |  |
| --- | --- |
| 1 | SELECT Item AS item\_description FROM Orders |

**29. Always and Everywhere!**

Wouldn’t it be great if there were a set of conditions you could depend on every time? The SQL queries using **ANY** and **ALL** can make this ideal a reality! Let’s look at how the **ALL** keyword is used to include records only when a set of conditions is true for **ALL** records. In the following example, we will return records from the Orders table where the idea is to get a list of high volume orders for a given item, in this case for customers who ordered more than 50 of the product:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Item FROM Orders  WHERE id = ALL  (SELECT ID FROM Orders  WHERE quantity > 50) |

**30. Writing Developer Friendly SQL**

An often overlooked but very important element of SQL scripting is adding comments to a script of queries to explain what it’s doing for the benefit of future developers who may need to revise and update your queries.  
The **—** single line and the **/\*** .. **\*/** multi-line delimiters empower us to add useful comments to scripts, but this is also used in another valuable way. Sometimes a section of code may not be in use, but we don’t want to delete it, because we anticipate using it again. Here we can simply add the comment delimiter to deactivate it momentarily:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | /\* This query below is commented so it won't execute\*/  /\*  SELECT item FROM Orders  WHERE date ALL = (SELECT Order\_ID FROM Orders  WHERE quantity > 50)  \*/    /\* the sql query below the will be executed  ignoring the text after "--"  \*/    SELECT item -- single comment  FROM Orders -- another single comment  WHERE id  ALL = (SELECT ID FROM Orders  WHERE quantity > 25) |

**31.  SQL queries for Database Management**

So far we have explored SQL query commands for querying tables and combining records from multiple queries. Now it’s time to take a step upward and look at the database on a structural level. Let’s start with the easiest SQL statement of all which creates a new database. Here, we are going to create the DB as a container for our Customers and Orders tables used in the previous ten examples above:

|  |  |
| --- | --- |
| 1 | CREATE DATABASE AllSales |

**32.**[Adding Tables to Our New DB](https://bytescout.com/blog/deep-sql-queries-and-examples.html#32)

Next, we will actually add the Customers table which we’ve been using in previous examples, and then add some of the column labels which we are already familiar with:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE Customers (  ID varchar(80),  Name varchar(80),  Phone varchar(20),  ....  ); |

Although most databases are created using a UI such as Access or OpenOffice, it is important to know how to create and delete databases and tables programmatically via code with SQL statements. This is especially so when installing a new web app and the UI asks new users to enter names for DBs to be added during installation.

**33.**[Modifying and Deleting Tables with SQL](https://bytescout.com/blog/deep-sql-queries-and-examples.html#33)

Imagine that you decide to send a birthday card to your customers to show your appreciation for their business, and so you want to add a birthday field to the Customers table. In these SQL examples, you see how easy it is to **modify existing tables with the ALTER statement:**

|  |  |
| --- | --- |
| 1 | ALTER TABLE Customers ADD Birthday varchar(80) |

If a table becomes corrupted with bad data you can quickly delete it like this:

|  |  |
| --- | --- |
| 1 | DROP TABLE table\_name |

**34.**[The Key to Successful Indexing](https://bytescout.com/blog/deep-sql-queries-and-examples.html#34)

Accurate indexing requires that the Primary Key column contain only unique values for this purpose. This guarantees that JOIN statements will maintain integrity and produce valid matches. Let’s create our Customers table again and establish the ID column as the Primary Key:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE Customers (  ID int NOT NULL,  Name varchar(80) NOT NULL,  PRIMARY KEY (ID)  ); |

We can extend the functionality of the Primary Key so that it automatically increments from a base. Change the ID entry above to add the *AUTO\_INCREMENT* keyword as in the following statement:

|  |  |
| --- | --- |
| 1 | ID int NOT NULL AUTO\_INCREMENT |

**35. Advanced Concepts For Improving Performance**

Whenever practical, is always better to write the column name list into a SELECT statement rather than using the **\***delimiter as a wildcard to select all columns. SQL Server has to do a search and replace operation to find all the columns in your table and write them into the statement for you (every time the SELECT is executed). For example:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Customers |

Would actually execute much faster on our database as:

|  |  |
| --- | --- |
| 1  2 | SELECT Name, Birthday, Phone,  Address, Zip FROM Customers |

Performance pitfalls can be avoided in many ways. For example, avoid the time sinkhole of forcing SQL Server to check the system/master database every time by using only a stored procedure name, and never prefix it with SP\_. Also setting NOCOUNT ON reduces the time required for SQL Server to count rows affected by INSERT, DELETE, and other commands. Using INNER JOIN with a condition is much faster than using WHERE clauses with conditions. We advise developers to learn SQL server queries to an advanced level for this purpose. For production purposes, these tips may be crucial to adequate performance. Notice that our tutorial examples tend to favor the INNER JOIN.

**36. Conditional Subquery Results**

The SQL operator EXISTS tests for the existence of records in a subquery and returns a value TRUE if a subquery returns one or more records. Have a look at this query with a subquery condition:

|  |  |
| --- | --- |
| 1  2  3 | SELECT Name FROM Customers WHERE EXISTS  (SELECT Item FROM Orders  WHERE Customers.ID = Orders.ID AND Price < 50) |

In this example above, the SELECT returns a value of TRUE when a customer has orders valued at less than $50.

**37. Copying Selections from Table to Table**

There are a hundred and one uses for this SQL tool. Suppose you want to archive your yearly Orders table into a larger archive table. This next example shows how to do it.

|  |  |
| --- | --- |
| 1  2  3 | INSERT INTO Yearly\_Orders  SELECT \* FROM Orders  WHERE Date<=1/1/2018 |

This example will add any records from the year 2018 to the archive.

**38. Catching NULL Results**

In cases where NULL values are allowed in a field, calculations on those values will produce NULL results as well. This can be avoided by use of the IFNULL operator. In this next example, a value of zero is returned rather than a value of NULL when the calculation encounters a field with NULL value:

|  |  |
| --- | --- |
| 1  2  3 | SELECT Item, Price \*  (QtyInStock + IFNULL(QtyOnOrder, 0))  FROM Orders |

**39. HAVING can be Relieving!**

The problem was that the SQL WHERE clause could not operate on aggregate functions. The problem was solved by using the HAVING clause. As an example, this next query fetches a list of customers by the region where there is at least one customer per region:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT COUNT(ID), Region  FROM Customers  GROUP BY Region  HAVING COUNT(ID) > 0; |

**40. Tie things up with Strings!**

Let’s have a look at processing the contents of field data using functions. Substring is probably the most valuable of all built-in functions. It gives you some of the power of Regex, but it’s not so complicated as Regex. Suppose you want to find the substring left of the dots in a web address. **Here’s how to do it with an SQL Select query**:

|  |  |
| --- | --- |
| 1 | SELECT SUBSTRING\_INDEX("www.bytescout.com", ".", 2); |

SQL, **S**tructured **Q**uery **L**anguage, is a programming language designed to manage data stored in relational databases. SQL operates through simple, declarative statements. This keeps data accurate and secure, and it helps maintain the integrity of databases, regardless of size.

Here’s an appendix of commonly used commands.

**COMMANDS**

**ALTER TABLE**

ALTER TABLE table\_name

ADD column\_name datatype;

ALTER TABLE lets you add columns to a table in a database.

**AND**

SELECT column\_name(s)

FROM table\_name

WHERE column\_1 = value\_1

AND column\_2 = value\_2;

AND is an operator that combines two conditions. Both conditions must be true for the row to be included in the result set.

**AS**

SELECT column\_name AS 'Alias'

FROM table\_name;

AS is a keyword in SQL that allows you to rename a column or table using an *alias*.

**AVG()**

SELECT AVG(column\_name)

FROM table\_name;

AVG() is an aggregate function that returns the average value for a numeric column.

**BETWEEN**

SELECT column\_name(s)

FROM table\_name

WHERE column\_name BETWEEN value\_1 AND value\_2;

The BETWEEN operator is used to filter the result set within a certain range. The values can be numbers, text or dates.

**CASE**

SELECT column\_name,

CASE

WHEN condition THEN 'Result\_1'

WHEN condition THEN 'Result\_2'

ELSE 'Result\_3'

END

FROM table\_name;

CASE statements are used to create different outputs (usually in the SELECT statement). It is SQL’s way of handling if-then logic.

**COUNT()**

SELECT COUNT(column\_name)

FROM table\_name;

COUNT() is a function that takes the name of a column as an argument and counts the number of rows where the column is not NULL.

**CREATE TABLE**

CREATE TABLE table\_name (

column\_1 datatype,

column\_2 datatype,

column\_3 datatype

);

CREATE TABLE creates a new table in the database. It allows you to specify the name of the table and the name of each column in the table.

**DELETE**

DELETE FROM table\_name

WHERE some\_column = some\_value;

DELETE statements are used to remove rows from a table.

**GROUP BY**

SELECT column\_name, COUNT(\*)

FROM table\_name

GROUP BY column\_name;

GROUP BY is a clause in SQL that is only used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into groups.

**HAVING**

SELECT column\_name, COUNT(\*)

FROM table\_name

GROUP BY column\_name

HAVING COUNT(\*) > value;

HAVING was added to SQL because the WHERE keyword could not be used with aggregate functions.

**INNER JOIN**

SELECT column\_name(s)

FROM table\_1

JOIN table\_2

ON table\_1.column\_name = table\_2.column\_name;

An inner join will combine rows from different tables if the *join condition* is true.

**INSERT**

INSERT INTO table\_name (column\_1, column\_2, column\_3)

VALUES (value\_1, 'value\_2', value\_3);

INSERT statements are used to add a new row to a table.

**IS NULL / IS NOT NULL**

SELECT column\_name(s)

FROM table\_name

WHERE column\_name IS NULL;

IS NULL and IS NOT NULL are operators used with the WHERE clause to test for empty values.

**LIKE**

SELECT column\_name(s)

FROM table\_name

WHERE column\_name LIKE pattern;

LIKE is a special operator used with the WHERE clause to search for a specific pattern in a column.

**LIMIT**

SELECT column\_name(s)

FROM table\_name

LIMIT number;

LIMIT is a clause that lets you specify the maximum number of rows the result set will have.

**MAX()**

SELECT MAX(column\_name)

FROM table\_name;

MAX() is a function that takes the name of a column as an argument and returns the largest value in that column.

**MIN()**

SELECT MIN(column\_name)

FROM table\_name;

MIN() is a function that takes the name of a column as an argument and returns the smallest value in that column.

**OR**

SELECT column\_name

FROM table\_name

WHERE column\_name = value\_1

OR column\_name = value\_2;

OR is an operator that filters the result set to only include rows where either condition is true.

**ORDER BY**

SELECT column\_name

FROM table\_name

ORDER BY column\_name ASC | DESC;

ORDER BY is a clause that indicates you want to sort the result set by a particular column either alphabetically or numerically.

**OUTER JOIN**

SELECT column\_name(s)

FROM table\_1

LEFT JOIN table\_2

ON table\_1.column\_name = table\_2.column\_name;

An outer join will combine rows from different tables even if the join condition is not met. Every row in the *left* table is returned in the result set, and if the join condition is not met, then NULL values are used to fill in the columns from the *right* table.

**ROUND()**

SELECT ROUND(column\_name, integer)

FROM table\_name;

ROUND() is a function that takes a column name and an integer as arguments. It rounds the values in the column to the number of decimal places specified by the integer.

**SELECT**

SELECT column\_name

FROM table\_name;

SELECT statements are used to fetch data from a database. Every query will begin with SELECT.

**SELECT DISTINCT**

SELECT DISTINCT column\_name

FROM table\_name;

SELECT DISTINCT specifies that the statement is going to be a query that returns unique values in the specified column(s).

**SUM**

SELECT SUM(column\_name)

FROM table\_name;

SUM() is a function that takes the name of a column as an argument and returns the sum of all the values in that column.

**UPDATE**

UPDATE table\_name

SET some\_column = some\_value

WHERE some\_column = some\_value;

UPDATE statements allow you to edit rows in a table.

**WHERE**

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator value;

WHERE is a clause that indicates you want to filter the result set to include only rows where the following *condition* is true.

**WITH**

WITH temporary\_name AS (

SELECT \*

FROM table\_name)

SELECT \*

FROM temporary\_name

WHERE column\_name operator value;

WITH clause lets you store the result of a query in a temporary table using an alias. You can also define multiple temporary tables using a comma and with one instance of the WITH keyword.