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

These are perhaps the most useful SQL query examples. In the example below, we are extracting the “Student\_ID” column or attribute from the table “STUDENT”. The select statement is used to select data from the database.

|  |  |
| --- | --- |
| 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?utm_referer=www.google.com#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 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. The order by clause is used to sort the data from the table. The order by clause should always be used in the last of the SQL query.

|  |  |
| --- | --- |
| 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.

Ascending (ASC) is the default condition for the ORDER BY clause. In other words, if users don’t specify ASC or DESC after the column name, then the result will be ordered in ascending order only.

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

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

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

The SQL query below will select Name, Age columns from the Patients table, then will filter them by Age value to include records where Age is more than 40 and then will group records with similar Age value and then finally will output them sorted by Name. The basic rule is that the group by clause should always follow a where clause in a Select statement and must precede the Order by clause.

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

Another sample of use of Group By: this expression will select records with a price lesser than 70 from the Orders table, will group records with a similar price, will sort the output by price, and will also add the column COUNT(price) that will display how many records with similar price were found:

|  |  |
| --- | --- |
| 1  2 | **SELECT** COUNT(price), price **FROM** orders  **WHERE** price < 70 **GROUP** **BY** price **ORDER** **BY** price |

Note: you should use the very same set of columns for both SELECT and GROUP BY commands, otherwise you will get an error. Many thanks to *Sachidannad* for pointing it out!

## 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. **In count, if users define DISTINCT, then they cal also define the query\_partition\_clause.** This clause is a part of the analytic clause, and other clauses such as order\_by\_clause and windowing\_clause are not permitted.

**Syntax: SELECT COUNT(colname) FROM table name;**

|  |  |
| --- | --- |
| 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?utm_referer=www.google.com#7)

SUM calculates the total of the attribute that is given to it as an argument. SUM is an aggregate function and it calculates the sum of all the distinct values. and the sum of all the duplicate values.

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

## 8. Data Manipulation Using AVG

Simple – an average of a given attribute. Average is also an aggregate function in SQL. The AVG() function computes the average of non-NULL values in a column. It ignores the null values.

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

## 9.   SQL 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. The view is also called a virtual table. In other words, a view is just a mirrored copy of a table whose data is the result of a stored query.

A view is a legitimate copy of a different table or sequence of tables. A view obtains its information or data from the tables from previously created tables known as base tables. Base tables are real tables. All procedures implemented on a view really modify the base table. Users can use views just like the real or base tables. In view, users can apply various DDL, DML commands such as update, insert into, and delete.

|  |  |
| --- | --- |
| 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’. The important thing to remember here is that the DROP VIEW is disallowed if there are any views dependent on the view you are about to drop.

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

## 14. Query to Display User Tables

**A user-defined table** is a representation of defined information in a table, and it 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. User tables explain the relational tables of the current user.

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

## 15. Query to Display Primary Keys

A primary key uniquely identifies all values within a table. A primary key imposes a NOT NULL restriction and a unique constraint in one declaration. In other words, it prevents various rows from having similar values or sequences of columns. It doesn’t allow null values. The primary key can be defined as a single column or the combination of two columns in a table. It is responsible for all the relationships between the tables.

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. A unique key also recognizes a different tuple uniquely in relation to or table. A table can have more than one unique key. Unique key constraints can take only one NULL value for the column.

|  |  |
| --- | --- |
| 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' |

Primary, Unique, and Foreign are part of the constraints in SQL. Constraints are essential to the scalability, compliance, and sincerity of the data. Constraints implement particular rules, assuring the data adheres to the conditions outlined. For example, these are the laws imposed on the columns of the database tables. These are applied to restrict the kind of data in the table. This assures the efficiency and authenticity of the database.

## 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 advanced 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' |

## 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 that are easily visualized. Our practice DB will include a Customer 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 that 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](https://bytescout.com/blog/postgresql-advanced-queries.html?utm_referer=www.google.com):

|  |  |
| --- | --- |
| 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?utm_referer=www.google.com#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 contains 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](https://bytescout.com/blog/postgresql-advanced-queries.html?utm_referer=www.google.com) 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?utm_referer=www.google.com#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 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 advanced SQL commands.

## 27. Doubling the Power with UNION

We can combine the results of two SQL query 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 a very powerful new table generation potential.

## 28. [Making Column Labels More Friendly](https://bytescout.com/blog/deep-sql-queries-and-examples.html?utm_referer=www.google.com#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 complex 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 SQL queries.

**A SQL script is a collection of SQL elements and commands accumulated as a file in SQL Scripts. This script file can include many SQL commands or PL/SQL codes. One can utilize SQL Scripts to build, edit, design, execute, and delete files.**

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 examples 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?utm_referer=www.google.com#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?utm_referer=www.google.com#33)

The ALTER statement is used to modify or change the meaning of a table. In the case of the relational tables with columns, ALTER statement is used to update the table to the new or modified rules or definition. **Alter** belongs to the DDL category of Commands. Data definition language can be described as a pattern for commands through which data structures are represented.

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](https://bytescout.com/blog/postgresql-advanced-queries.html?utm_referer=www.google.com), 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?utm_referer=www.google.com#34)

An index is a schema element that includes a record for each content that arrives in the indexed column of the database table or cluster and gives a high-speed path to rows. There are many types of indexes such as Bitmap indexes, Partitioned indexes, Function-based indexes, and Domain indexes.

Accurate indexing requires that the Primary Key column contains 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](https://bytescout.com/blog/postgresql-advanced-queries.html?utm_referer=www.google.com) 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

The NULL is the terminology applied to describe an absent value. Null does not mean zero. A NULL value in a column of a table is a condition in a domain that seems to be empty. A column with a NULL value is a domain with an absent value. It is essential to recognize that a NULL value is distinct from a zero.

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 the 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 a 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); |

This line will return everything to the left of the second occurrence of “. ” and so, in this case, it will return

|  |  |
| --- | --- |
| 1 | <**a** href="[https://bytescout.com](https://bytescout.com/)">www.bytescout.com</**a**> |

**Check this video to learn about every SQL query:**

**.. and 20 more useful SQL Queries examples!!**

## 41. Use COALESCE to return the first non-null expression

The SQL Coalesce is used to manage the NULL values of the database. In this method, the NULL values are substituted with the user-defined value. The SQL Coalesce function assesses the parameters in series and always delivers the first non-null value from the specified argument record.

### Syntax

|  |  |
| --- | --- |
| 1 | **SELECT** COALESCE(NULL,NULL,'ByteScout',NULL,'Byte') |

### Output

ByteScout

## 42. Use Convert to transform any value into a particular datatype

This is used to convert a value into a defined datatype. For example, if you want to convert a particular value into int datatype then a convert function can be used to achieve this. For example,

### Syntax

|  |  |
| --- | --- |
| 1 | **SELECT** CONVERT(**int**, 27.64) |

### Output

27

## 43. DENSE\_RANK()Analytical query

It is an analytic query that computes the rank of a row in an arranged collection of rows. An output rank is a number starting from 1. DENSE\_RANK is one of the most important SQL queries. It returns rank preferences as sequential numbers. It does not jump rank in event of relations. For example, the following query will give the sequential ranks to the employee.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **SELECT** eno,  dno,  salary,  DENSE\_RANK() OVER (PARTITION **BY** dno **ORDER** **BY** salary) **AS** ranking  **FROM** employee;    ENO  DNO SALARY RANKING  ---------- ---------- ---------- ----------  7933  10 1500   1  7788  10 2650   2  7831  10 6000   3  7362  20 900    1  7870  20 1200   2  7564  20 2575   3  7784  20 4000   4  7903  20 4000   4  7901  30 550    1  7655  30 1450   2  7522  30 1450   2  7844  30 1700   3  7493  30 1500   4  7698  30 2850   5 |

## 44. Query\_partition\_clause

The query\_partition\_clause breaks the output set into distributions, or collections, of data. The development of the analytic query is limited to the confines forced by these partitions, related to the process a GROUP BY clause modifies the performance of an aggregate function. If the query\_partition\_clause is eliminated, the entire output collection is interpreted as a separate partition.

**The following query applies an OVER clause, so the average displayed is based on all the records of the output set.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **SELECT** eno, dno, salary,  AVG(salary) OVER () **AS** avg\_sal  **FROM** employee;    EO  DNO SALARY AVG\_SAL  ---------- ---------- ---------- ----------  7364 20 900 2173.21428  7494 30 1700 2173.21428  7522 30 1350 2173.21428  7567 20 3075 2173.21428  7652 30 1350 2173.21428  7699 30 2950 2173.21428  7783 10 2550 2173.21428  7789 20 3100 2173.21428  7838 10 5100 2173.21428  7845 30 1600 2173.21428  7877 20 1200 2173.21428  7901 30 1050 2173.21428  7903 20 3100 2173.21428  7935 10 1400 2173.21428 |

## 45. Finding the last five records from the table

Now, if you want to fetch the last eight records from the table then it is always difficult to get such data if your table contains huge information. For example, you want to get the last 8 records from the employee table then you can use rownum and a union clause. The rownum is temporary in SQL.

For example,

|  |  |
| --- | --- |
| 1  2  3 | **Select** \* **from** Employee A **where** rownum <=8  **union**  **select** \* **from** (**Select** \* **from** Employee A **order** **by** rowid **desc**) **where** rownum <=8; |

**The above SQL query will give you the last eight records from the employee table where rownum is a pseudo column. It indexes the data in an output set.**

## 46. LAG

The LAG is applied to get data from a prior row. This is an analytical function. For example, the following query gives the salary from the prior row to compute the difference between the salary of the current row and that of the prior row. **In this query, the ORDER BY of the LAG function is applied. The default is 1 if you do not define offset.** The arbitrary default condition is given if the offset moves past the range of the window. The default is null if you do not define default.

### Syntax

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **SELECT** dtno,         eno,         emname,         job,         salary,         LAG(sal, 1, 0) OVER (PARTITION **BY** dtno **ORDER** **BY** salary) **AS** salary\_prev  **FROM**   employee; |

### Output

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | DTNO ENO ENAME JOB SAL SAL\_PREV  ---------- ---------- ---------- --------- ---------- ----------  10 7931 STEVE CLERK 1300 0  10 7783 JOHN MANAGER 2450 1300  10 7834 KING PRESIDENT 5000 2450  20 7364 ROBIN CLERK 800 0  20 7876 BRIAN CLERK 1100 800  20 7567 SHANE MANAGER 2975 1100  20 7784 SCOTT ANALYST 3000 2975  20 7908 KANE ANALYST 3000 3000  30 7900 JAMES CLERK 950 0  30 7651 CONNER SALESMAN 1250 950  30 7522 MATTHEW SALESMAN 1250 1250  30 7843 VIVIAN SALESMAN 1500 1250  30 7494 ALLEN SALESMAN 1600 1500  30 7695 GLEN MANAGER 2850 1600 |

## 47. LEAD

The LEAD is also an analytical query that is applied to get data from rows extra down the output set. The following query gives the salary from the next row to compute the deviation between the salary of the prevailing row and the subsequent row. The default is 1 if you do not define offset. The arbitrary default condition is given if the offset moves past the range of the window. The default is null if you do not define default.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **SELECT** eno,         empname,         job,         salary,         LEAD(salary, 1, 0) OVER (**ORDER** **BY** salary) **AS** salary\_next,         LEAD(salary, 1, 0) OVER (**ORDER** **BY** salary) - salary **AS** salary\_diff  **FROM**   employee;    ENO EMPNAME JOB SALARY SALARY\_NEXT SALARY\_DIFF  ---------- ---------- --------- ---------- ---------- ----------  7369 STEVE CLERK 800 950 150  7900 JEFF CLERK 950 1100 150  7876 ADAMS CLERK 1100 1250 150  7521 JOHN SALESMAN 1250 1250 0  7654 MARK SALESMAN 1250 1300 50  7934 TANTO CLERK 1300 1500 200  7844 MATT SALESMAN 1500 1600 100  7499 ALEX SALESMAN 1600 2450 850  7782 BOON MANAGER 2450 2850 400  7698 BLAKE MANAGER 2850 2975 125  7566 JONES MANAGER 2975 3000 25  7788 SCOTT ANALYST 3000 3000 0  7902 FORD ANALYST 3000 5000 2000  7839 KING PRESIDENT 5000 0 -5000 |

## 48. PERCENT\_RANK

The PERCENT\_RANK analytic query. The ORDER BY clause is necessary for this query. Excluding a partitioning clause from the OVER clause determines the entire output set is interpreted as a separate partition. **The first row of the standardized set is indicated 0 and the last row of the set is indicated 1.** For example, the SQL query example gives the following output.

### Syntax

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **SELECT**       prdid, SUM(amount),       PERCENT\_RANK() OVER (**ORDER** **BY** SUM(amount) **DESC**) **AS** percent\_rank  **FROM** sales  **GROUP** **BY** prdid  **ORDER** **BY** prdid; |

### Output

|  |  |
| --- | --- |
| 1  2  3  4 | PRDID        SUM(AMOUNT)  PERCENT\_RANK  ----------- ----------- ------------            1    22623.5            0            2   223927.08           1 |

## 49. MIN

Utilizing a blank OVER clause converts the MIN into an analytic function. This is also an analytical query. In this, the entire result set is interpreted as a single partition. It gives you the minimum salary for all employees and their original data. For example, the following query is displaying the use of MIN in the Select query.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **SELECT** eno,         empname,         dtno,         salary,  **MIN**(salary) OVER (PARTITION **BY** dtno) **AS** min\_result  **FROM**   employee;         ENO   EMPNAME          DTNO     SALARY MIN\_RESULT  ---------- ---------- ---------- ---------- ---------------        7782 CLARK              10       2450            1300        7839 KING               10       5000            1300        7934 MILLER             10       1300            1300        7566 JONES              20       2975             800        7902 FORD               20       3000             800        7876 ADAMS              20       1100             800        7369 SMITH              20        800             800        7788 SCOTT              20       3000             800        7521 WARD               30       1250             950        7844 TURNER             30       1500             950        7499 ALLEN              30       1600             950        7900 JAMES              30        950             950        7698 BLAKE              30       2850             950        7654 MARTIN             30       1250             950 |

## 50. MAX

Using a blank row OVER clause converts the MAX into an analytic function. **The lack of a partitioning clause indicates the entire output set is interpreted as a separate partition.** This gives the maximum salary for all employees and their original data. For example, the following query displays the use of MAX in the select query.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **SELECT** eno,         empname,         dtno,         salary,  **MAX**(salary) OVER () **AS** max\_result  **FROM**   employee;         ENO   EMPNAME          DTNO       SALARY    MAX\_RESULT  ---------- ---------- ---------- ---------- ----------        7369 SMITH              20        800       3000        7499 ALLEN              30       1600       3000        7521 WARD               30       1250       3000        7566 JONES              20       2975       3000        7654 MARTIN             30       1250       3000        7698 BLAKE              30       2850       3000        7782 CLARK              10       2450       3000        7788 SCOTT              20       3000       3000        7839 KING               10       5000       3000        7844 TURNER             30       1500       3000        7876 ADAMS              20       1100       3000        7900 JAMES              30        950       3000        7902 FORD               20       3000       3000        7934 MILLER             10       1300       3000 |

## 51. Top- N queries

Top-N queries give a process for restricting the number of rows delivered from organized assemblages of data. They are remarkably beneficial when users want to give the top or bottom number of rows from a table.

**For example, the following query gives the 20 rows with 10 different values:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | SELECT price  FROM   sales\_order  ORDER BY price;    PRICE  ----------  100  100  200  200  300  300  400  400  500  500  600    PRICE  ----------  600  700  700  800  800  900  900  1000  1000    20 rows selected. |

## 52. CORR Analytic Query

The CORR analytic function is utilized to determine the coefficient of correlation. This query is also used to calculate the Pearson correlation coefficient. The function calculates the following on rows in the table with no null values. This query always returns the values between +1 and -1, which describe the following:

Syntax: CORR(exp1, exp2) [ OVER (analytic\_clause) ]

### Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT empid,         name,         dno,         salary,         job,         CORR(SYSDATE - joiningdate, salary) OVER () AS my\_corr\_val  FROM   employee; |

## 53. NTILE Analytic Query

The NTILE enables users to split a sequence set into a detailed number of relatively similar groups, or containers, rows sanctioning. If the number of rows in the collection is less than the number of containers defined, the number of containers will be decreased. The basic syntax is as displayed below:

NTILE(exp) OVER ([ partition\_clause ] order\_by)

### Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT empid,         name,         dno,         salary,         NTILE(6) OVER (ORDER BY salary) AS container\_no  FROM   employee; |

## 54. VARIANCE, VAR\_POP, and VAR\_SAMP Query

The VARIANCE, VAR\_POP, and VAR\_SAMP are aggregate functions. These are utilized to determine the variance, group variance, and sample variance of a collection of data individually. As aggregate queries or functions, they decrease the number of rows, therefore the expression “aggregate”. If the data isn’t arranged we change the total rows in the Employee table to a separate row with the aggregated values. For example, the following query is displaying the use of these functions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT VARIANCE(salary) AS var\_salary,         VAR\_POP(salary) AS pop\_salary,         VAR\_SAMP(salary) AS samp\_salary  FROM   employee;    VAR\_SALARY   POP\_SALARY   SAMP\_SALARY  ------------ ----------- ------------  1479414.97  1588574.81   1388717.27 |

## 55. STDDEV, STDDEV\_POP, and STDDEV\_SAMP Queries

The STDDEV, STDDEV\_POP, and STDDEV\_SAMP aggregate queries or functions are applied to determine the standard deviation, population standard deviation, and cumulative sample standard deviation individually. As aggregate queries, they decrease the number of rows, therefore the expression “aggregate”. If the data isn’t arranged we convert all the rows in the EMPLOYEE table to a separate row. For example, the following query is displaying the use of all these functions.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT STDDEV(salary) AS stddev\_salary,         STDDEV\_POP(salary) AS pop\_salary,         STDDEV\_SAMP(salary) AS samp\_salary  FROM   employee;    STDDEV\_SALARY POP\_SALARY SAMP\_SALARY  ---------- -------------- ---------------  1193.50     1159.588      1193.603 |

If there is more than one account after dropping nulls, the STDDEV function gives the result of the STDDEV\_SAMP. Using an empty OVER clause converts the STDDEV query result into an analytic query. The absence of a partitioning indicates the entire output set is interpreted as a particular partition, so we accept the standard deviation of the salary and the primary data.

## 56. Pattern Matching

The pattern matching syntax adds various alternatives. Data must be treated precisely and in a proper form. The PARTITION BY and ORDER BY conditions of all SQL analytic queries is applied to split the data into accumulations and within each group. If no partitions are specified, it is considered the entire sequence set is one huge partition.

For example,

**The MEASURES clause specifies the column result that will be provided for each match.**

### Syntax

|  |  |
| --- | --- |
| 1  2  3 | MEASURES  STRT.tstamp AS initial\_tstamp,            LAST(UP.tstamp) AS first\_tstamp,            LAST(DOWN.tstamp) AS finished\_tstamp |

### Example

|  |  |
| --- | --- |
| 1  2  3  4 | DEFINE    UP AS UP.products\_sold > PREV(UP.products\_sold),    FLAT AS FLAT.products\_sold = PREV(FLAT.products\_sold),    DOWN AS DOWN.products\_sold < PREV(DOWN.products\_sold) |

## 57. FIRST\_VALUE

The simplest way to get analytic functions is to begin by studying aggregate functions. An aggregate function collects or gathers data from numerous rows into a unique result row. For instance, users might apply the AVG function to get an average of all the salaries in the EMPLOYEE table. Let’s take a look at how First\_Value can be used. The primary explanation for the FIRST\_VALUE analytic function is displayed below.

### Syntax:

|  |  |
| --- | --- |
| 1  2  3  4  5 | FIRST\_VALUE    { (expr) [NULLS ]    | (expr [NULLS ])    }    OVER (analytic clause) |

### Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT eno,         dno,         salary,         FIRST\_VALUE(salary) IGNORE NULLS           OVER (PARTITION BY dno ORDER BY salary) AS lowest\_salary\_in\_dept  FROM   employee; |

The above query will ignore null values.

## 58. LAST\_VALUE

The primary explanation for the LAST\_VALUE analytic query or function is displayed below.

|  |  |
| --- | --- |
| 1  2  3  4 | Syntax: LAST\_VALUE    { (expr) [ { NULLS ]    | (expr [ NULLS ])    OVER (analytic clause) |

The LAST\_VALUE analytic query is related to the LAST analytic function. The function enables users to get the last output from an organized column. Applying the default windowing to the output can be surprising. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT eno,         dno,         salary,         LAST\_VALUE(salary) IGNORE NULLS           OVER (PARTITION BY dno ORDER BY salary) AS highest\_salary\_in\_dept  FROM   employee; |

## 59. Prediction

The design sample foretells the gender and age of clients who are most expected to adopt an agreement card (target = 1). The PREDICTION function takes the price matrix correlated with the design and applies for marital status, and house size as predictors. The syntax of the PREDICTION function can also apply a piece of arbitrary GROUPING information when getting a partitioned model.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | SELECT client\_gender, COUNT(\*) AS ct, ROUND(AVG(age)) AS average\_age     FROM mining\_data\_shop     WHERE PREDICTION(sample COST MODEL        USING client\_marital\_status, house\_size) = 1     GROUP BY client\_gender     ORDER BY client\_gender;    CUST\_GENDER         CNT    AVG\_AGE  ------------ ---------- ----------  F                   270         40  M                   585         41 |

## 60. CLUSTER\_SET

CLUSTER\_SET can get the data in one of the couple steps: It can use a mining type object to the information, or it can mine the data by performing an analytic clause that creates and uses one or more moving mining patterns.

This example enumerates the properties that have the biggest influence on cluster distribution for client ID 1000. The query requests the CLUSTER\_DETAILS and CLUSTER\_SET functions, which use the clustering model my\_sample.

### Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT S.cluster\_id, prob,         CLUSTER\_DETAILS(my\_sample, S.cluster\_id, 7 USING T.\*) kset  FROM    (SELECT v.\*, CLUSTER\_SET(my\_sample, USING \*) nset      FROM mining\_data     WHERE client\_id = 1000) T,    TABLE(T.nset) Q  ORDER BY 2 DESC; |

A cluster is a group table that distributes the corresponding data blocks i.e. all the tables are actually put together. For example, EMPLOYEE and DEPARTMENT tables are connected to the DNO column. If you cluster them, it will actually store all rows in the same data blocks.

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

## 61. WITH (Common Table Expressions)

A common table expression (CTE) is a defined short result set that endures within the range of a particular statement and that can be called later within that statement, perhaps on many occasions. The following query is describing the CTE:

### Syntax

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | WITH all\_emp  AS  (  SELECT empId, BossId, FirstName, LastName  FROM Emp  WHERE BossId is NULL    UNION ALL    SELECT e.empId, e.BossId, e.FirstName, e.LastName  FROM Emp e INNER JOIN all\_emp r  ON e.BossId = r.Id  )  SELECT \* FROM all\_emp |

## 62. NANVL

This function is utilized to deliver an optional value n1 if the inserted value n2 is NaN (not a number), and gives n2 if n2 is not a number. This function is used only for type BINARY\_FLOAT. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1  2 | SELECT bin\_float, NANVL(bin\_float,0)  FROM my\_demo\_table; |

## 63. WIDTH\_BUCKET

This function is used to obtain the bucket number. In this, it gives the value of the expression that would come under after being assessed. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT emp\_id, first\_name,last\_name,dept\_id,mgr\_id,  WIDTH\_BUCKET(department\_id,20,40,10) "Exists in Dept"  FROM emp  WHERE mgr\_id < 300  ORDER BY "Exists in Dept"; |

## 64. COSH

This function is used to deliver the hyperbolic cosine of a number. It accepts all numeric or non-numeric data types as an argument. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1 | SELECT COSH(0) "COSH of 0" FROM DUAL; |

## 65. SOUNDEX

The SOUNDEX function delivers a character string comprising the description of char. It allows users to match words that are spelled antagonistically, but sound similar in English. It does not support CLOB. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT last\_name, first\_name  FROM hr.emp  WHERE SOUNDEX(last\_name)  = SOUNDEX('SCOTTY'); |

## 66. TZ\_OFFSET

The TZ\_OFFSET gives the time zone offset identical to the case based on the date the statement is given. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1 | SELECT TZ\_OFFSET('US/Eastern') FROM DUAL; |

## 67. CARDINALITY

CARDINALITY is utilized to obtain the number of components in a nested table. It is supported in different versions. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1  2 | SELECT product\_id, CARDINALITY(ad\_mydocs\_get)  FROM my\_media\_table; |

## 68. DUMP

DUMP is one of the important string/char functions. It is utilized to get a VARCHAR2 value. The value delivered defines the data type code. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1  2 | SELECT DUMP('pqr', 1033)  FROM DUAL; |

## 69. PATH

PATH is applied simply with the UNDER\_PATH and EQUALS\_PATH requirements. It gives the corresponding path that points to the resource defined in the main state. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1  2 | SELECT ANY\_PATH FROM RESOURCE\_VIEW  WHERE EQUALS\_PATH(res, '/sys/schemas/OE/www.pqr.com')=3; |

## 70. UNISTR

UNISTR accepts an expression that determines character data and delivers it in the general character set. It gives support to the Unicode string literals by allowing users to define the Unicode value. The following query is displaying its use:

### Example

|  |  |
| --- | --- |
| 1 | SELECT UNISTR('pqr\00e4\00f3\00f9') FROM DUAL; |

## Database Queries in SQL

These are some of the commonly used database queries in SQL.

1. SELECT – database query used to extract data from a table.
2. CREATE DATABASE – database query used to create a new database.
3. DROP DATABASE – database query used to delete a database.
4. CREATE TABLE – database query used to create a table in the specified database.
5. ALTER TABLE – database query used to modify an existing table in the specified database
6. DROP TABLE – database query used to delete an existing table in the specified database
7. CREATE INDEX – Index creation query.
8. CREATE VIEW – View creation query.
9. DROP VIEW – View deletion query.
10. CREATE PROCEDURE – Procedure creation query.
11. CREATE FUNCTION – Function creation query.
12. DROP PROCEDURE – Procedure deletion query.
13. DROP FUNCTION – Function deletion query.

## Example of Query in a Database

We will be looking at some SELECT examples of a query in a database, as this is the most common command.

SELECT \* FROM employeeTable WHERE emp\_no = ‘12’;

This query filters out results that do not match a specified search.

SELECT \* FROM sys.objects WHERE object\_id=object\_id(‘<table name>’);

This database query will list all the columns of the table whose name matches the argument.

## Database Queries Examples

The following are some database queries examples that deal with creating tables, in a bit more advanced fashion.

1. Create a table with a primary key called “ID”.

CREATE TABLE table\_name (

PRIMARY KEY(column\_name)

);

1. Create a table with a non-unique index called “IDX” on column\_name.

CREATE INDEX idx\_name ON table\_name (column\_name);

1. Create a view with the name “VIEW1” that can be used to query data from table1. The view is created on columns column1 and column2. It must return the same number of rows as the underlying table, and it must return the same data type. In this case, we will return the maximum value for each column in the underlying table when queried against the view. The following query will be used to populate our view:

CREATE VIEW view1 AS SELECT MAX(column1), MAX(column2) FROM table1;

## Common Database Queries

Some more common database queries you will need are:

1)    The Maximum Value for a Column.

SELECT MAX(column\_name) FROM table\_name

2)    The Minimum Value for a Column.

SELECT MIN(column\_name) FROM table\_name

3)    The Count of Rows in a Table.

SELECT COUNT(\*) FROM table\_name;

## DBMS Queries with Examples

DBMS stands for DataBase Management System. Following are some DBMS queries with examples; these are very important for developers and database admins alike.

**1.** The List of Values for a Column.

SELECT column\_name, column\_commalist FROM table\_name;

**2.** The Difference between two Column Values.

SELECT column\_name(+) FROM table\_name WHERE column\_name(+) != value;

III.         The List of Unique Values for a Column.

SELECT DISTINCT column\_name FROM table\_name;

[**https://www.techbeamers.com/sql-query-questions-answers-for-practice/**](https://www.techbeamers.com/sql-query-questions-answers-for-practice/)

**50 SQL Query Questions and Answers for Practice**

* Post author:By: [Meenakshi Agarwal](https://www.techbeamers.com/author/meenakshi/)
* Post category:[SQL Interview](https://www.techbeamers.com/sql-interview/)
* Post last modified:April 29, 2023

If you want to improve your SQL skills, then install a SQL package like MySQL and start practicing with it. To get you started, we’ve outlined a few SQL query questions in this post.

Solving practice questions is the fastest way to learn any subject. That’s why we’ve selected a set of 50 SQL queries that you can use to step up your learning.

Most of these SQL query questions we’ve filtered out of interviews held by top IT MNCs like Flipkart and Amazon.

We recommend that you first try to form queries by yourself rather than just reading them from the post.

This tutorial includes SQL scripts to create the test data. So, you can use them to create a test database and tables.

Once you have done enough SQL practice, then do also check out another post on [SQL interview prep for QA and DBAs](https://www.techbeamers.com/top-sql-interview-questions-dba-qa/).

**Let’s Begin Learning SQL!**

**Prepare Sample Data To Practice SQL Skills.**

**Sample Table – Worker**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **WORKER\_ID** | **FIRST\_NAME** | **LAST\_NAME** | **SALARY** | **JOINING\_DATE** | **DEPARTMENT** |
| 001 | Monika | Arora | 100000 | 2014-02-20 09:00:00 | HR |
| 002 | Niharika | Verma | 80000 | 2014-06-11 09:00:00 | Admin |
| 003 | Vishal | Singhal | 300000 | 2014-02-20 09:00:00 | HR |
| 004 | Amitabh | Singh | 500000 | 2014-02-20 09:00:00 | Admin |
| 005 | Vivek | Bhati | 500000 | 2014-06-11 09:00:00 | Admin |
| 006 | Vipul | Diwan | 200000 | 2014-06-11 09:00:00 | Account |
| 007 | Satish | Kumar | 75000 | 2014-01-20 09:00:00 | Account |
| 008 | Geetika | Chauhan | 90000 | 2014-04-11 09:00:00 | Admin |

**Sample Table – Bonus**

|  |  |  |
| --- | --- | --- |
| **WORKER\_REF\_ID** | **BONUS\_DATE** | **BONUS\_AMOUNT** |
| 1 | 2016-02-20 00:00:00 | 5000 |
| 2 | 2016-06-11 00:00:00 | 3000 |
| 3 | 2016-02-20 00:00:00 | 4000 |
| 1 | 2016-02-20 00:00:00 | 4500 |
| 2 | 2016-06-11 00:00:00 | 3500 |

**Sample Table – Title**

|  |  |  |
| --- | --- | --- |
| **WORKER\_REF\_ID** | **WORKER\_TITLE** | **AFFECTED\_FROM** |
| 1 | Manager | 2016-02-20 00:00:00 |
| 2 | Executive | 2016-06-11 00:00:00 |
| 8 | Executive | 2016-06-11 00:00:00 |
| 5 | Manager | 2016-06-11 00:00:00 |
| 4 | Asst. Manager | 2016-06-11 00:00:00 |
| 7 | Executive | 2016-06-11 00:00:00 |
| 6 | Lead | 2016-06-11 00:00:00 |
| 3 | Lead | 2016-06-11 00:00:00 |

To prepare the sample data, you can run the following queries in your database query executor or on the SQL command line. We’ve tested them with MySQL Server 5.7 and MySQL Workbench 6.3.8 query browser. You can also download these tools and install them to execute the SQL queries.

**SQL Script to Seed Sample Data.**

CREATE DATABASE ORG;

SHOW DATABASES;

USE ORG;

CREATE TABLE Worker (

WORKER\_ID INT NOT NULL PRIMARY KEY AUTO\_INCREMENT,

FIRST\_NAME CHAR(25),

LAST\_NAME CHAR(25),

SALARY INT(15),

JOINING\_DATE DATETIME,

DEPARTMENT CHAR(25)

);

INSERT INTO Worker

(WORKER\_ID, FIRST\_NAME, LAST\_NAME, SALARY, JOINING\_DATE, DEPARTMENT) VALUES

(001, 'Monika', 'Arora', 100000, '14-02-20 09.00.00', 'HR'),

(002, 'Niharika', 'Verma', 80000, '14-06-11 09.00.00', 'Admin'),

(003, 'Vishal', 'Singhal', 300000, '14-02-20 09.00.00', 'HR'),

(004, 'Amitabh', 'Singh', 500000, '14-02-20 09.00.00', 'Admin'),

(005, 'Vivek', 'Bhati', 500000, '14-06-11 09.00.00', 'Admin'),

(006, 'Vipul', 'Diwan', 200000, '14-06-11 09.00.00', 'Account'),

(007, 'Satish', 'Kumar', 75000, '14-01-20 09.00.00', 'Account'),

(008, 'Geetika', 'Chauhan', 90000, '14-04-11 09.00.00', 'Admin');

CREATE TABLE Bonus (

WORKER\_REF\_ID INT,

BONUS\_AMOUNT INT(10),

BONUS\_DATE DATETIME,

FOREIGN KEY (WORKER\_REF\_ID)

REFERENCES Worker(WORKER\_ID)

ON DELETE CASCADE

);

INSERT INTO Bonus

(WORKER\_REF\_ID, BONUS\_AMOUNT, BONUS\_DATE) VALUES

(001, 5000, '16-02-20'),

(002, 3000, '16-06-11'),

(003, 4000, '16-02-20'),

(001, 4500, '16-02-20'),

(002, 3500, '16-06-11');

Copy

CREATE TABLE Title (

WORKER\_REF\_ID INT,

WORKER\_TITLE CHAR(25),

AFFECTED\_FROM DATETIME,

FOREIGN KEY (WORKER\_REF\_ID)

REFERENCES Worker(WORKER\_ID)

ON DELETE CASCADE

);

INSERT INTO Title

(WORKER\_REF\_ID, WORKER\_TITLE, AFFECTED\_FROM) VALUES

(001, 'Manager', '2016-02-20 00:00:00'),

(002, 'Executive', '2016-06-11 00:00:00'),

(008, 'Executive', '2016-06-11 00:00:00'),

(005, 'Manager', '2016-06-11 00:00:00'),

(004, 'Asst. Manager', '2016-06-11 00:00:00'),

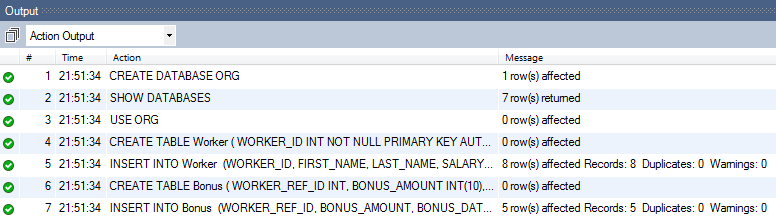
(007, 'Executive', '2016-06-11 00:00:00'),

(006, 'Lead', '2016-06-11 00:00:00'),

(003, 'Lead', '2016-06-11 00:00:00');

Copy

Once the above SQL would run, you’ll see a result similar to the one attached below.

**Creating Sample Data to Practice SQL Skill.**

**Practice with 50 challenging SQL query questions.**

**Q-1. Write an SQL query to fetch “FIRST\_NAME” from the Worker table using the alias name <WORKER\_NAME>.**

**Ans.**

The required query is:

Select FIRST\_NAME AS WORKER\_NAME from Worker;

Copy

**Q-2. Write an SQL query to fetch “FIRST\_NAME” from the Worker table in upper case.**

**Ans.**

The required query is:

Select upper(FIRST\_NAME) from Worker;

Copy

**Q-3. Write an SQL query to fetch unique values of DEPARTMENT from the Worker table.**

**Ans.**

The required query is:

Select distinct DEPARTMENT from Worker;

Copy

**Q-4. Write an SQL query to print the first three characters of  FIRST\_NAME from the Worker table.**

**Ans.**

The required query is:

Select substring(FIRST\_NAME,1,3) from Worker;

Copy

**Q-5. Write an SQL query to find the position of the alphabet (‘a’) in the first name column ‘Amitabh’ from the Worker table.**

**Ans.**

The required query is:

Select INSTR(FIRST\_NAME, BINARY'a') from Worker where FIRST\_NAME = 'Amitabh';

Copy

**Notes.**

* The INSTR does a case-insensitive search.
* Using the BINARY operator will make INSTR work as the case-sensitive function.

**Q-6. Write an SQL query to print the FIRST\_NAME from the Worker table after removing white spaces from the right side.**

**Ans.**

The required query is:

Select RTRIM(FIRST\_NAME) from Worker;

Copy

**Q-7. Write an SQL query to print the DEPARTMENT from the Worker table after removing white spaces from the left side.**

**Ans.**

The required query is:

Select LTRIM(DEPARTMENT) from Worker;

Copy

**Q-8. Write an SQL query that fetches the unique values of DEPARTMENT from the Worker table and prints its length.**

**Ans.**

The required query is:

Select distinct length(DEPARTMENT) from Worker;

Copy

**Q-9. Write an SQL query to print the FIRST\_NAME from the Worker table after replacing ‘a’ with ‘A’.**

**Ans.**

The required query is:

Select REPLACE(FIRST\_NAME,'a','A') from Worker;

Copy

**Q-10. Write an SQL query to print the FIRST\_NAME and LAST\_NAME from the Worker table into a single column COMPLETE\_NAME. A space char should separate them.**

**Ans.**

The required query is:

Select CONCAT(FIRST\_NAME, ' ', LAST\_NAME) AS 'COMPLETE\_NAME' from Worker;

Copy

**Q-11. Write an SQL query to print all Worker details from the Worker table order by FIRST\_NAME Ascending.**

**Ans.**

The required query is:

Select \* from Worker order by FIRST\_NAME asc;

Copy

**Q-12. Write an SQL query to print all Worker details from the Worker table order by FIRST\_NAME Ascending and DEPARTMENT Descending.**

**Ans.**

The required query is:

Select \* from Worker order by FIRST\_NAME asc,DEPARTMENT desc;

Copy

**Q-13. Write an SQL query to print details for Workers with the first names “Vipul” and “Satish” from the Worker table.**

**Ans.**

The required query is:

Select \* from Worker where FIRST\_NAME in ('Vipul','Satish');

Copy

**Q-14. Write an SQL query to print details of workers excluding first names, “Vipul” and “Satish” from the Worker table.**

**Ans.**

The required query is:

Select \* from Worker where FIRST\_NAME not in ('Vipul','Satish');

Copy

**Q-15. Write an SQL query to print details of Workers with DEPARTMENT name as “Admin”.**

**Ans.**

The required query is:

Select \* from Worker where DEPARTMENT like 'Admin%';

Copy

**Q-16. Write an SQL query to print details of the Workers whose FIRST\_NAME contains ‘a’.**

**Ans.**

The required query is:

Select \* from Worker where FIRST\_NAME like '%a%';

Copy

**Q-17. Write an SQL query to print details of the Workers whose FIRST\_NAME ends with ‘a’.**

**Ans.**

The required query is:

Select \* from Worker where FIRST\_NAME like '%a';

Copy

**Q-18. Write an SQL query to print details of the Workers whose FIRST\_NAME ends with ‘h’ and contains six alphabets.**

**Ans.**

The required query is:

Select \* from Worker where FIRST\_NAME like '\_\_\_\_\_h';

Copy

**Q-19. Write an SQL query to print details of the Workers whose SALARY lies between 100000 and 500000.**

**Ans.**

The required query is:

Select \* from Worker where SALARY between 100000 and 500000;

Copy

**Q-20. Write an SQL query to print details of the Workers who joined in Feb’2014.**

**Ans.**

The required query is:

Select \* from Worker where year(JOINING\_DATE) = 2014 and month(JOINING\_DATE) = 2;

Copy

**Q-21. Write an SQL query to fetch the count of employees working in the department ‘Admin’.**

**Ans.**

The required query is:

SELECT COUNT(\*) FROM worker WHERE DEPARTMENT = 'Admin';

Copy

**Q-22. Write an SQL query to fetch worker names with salaries >= 50000 and <= 100000.**

**Ans.**

The required query is:

SELECT CONCAT(FIRST\_NAME, ' ', LAST\_NAME) As Worker\_Name, Salary

FROM worker

WHERE WORKER\_ID IN

(SELECT WORKER\_ID FROM worker

WHERE Salary BETWEEN 50000 AND 100000);

Copy

**Q-23. Write an SQL query to fetch the no. of workers for each department in descending order.**

**Ans.**

The required query is:

SELECT DEPARTMENT, count(WORKER\_ID) No\_Of\_Workers

FROM worker

GROUP BY DEPARTMENT

ORDER BY No\_Of\_Workers DESC;

Copy

**Q-24. Write an SQL query to print details of the Workers who are also Managers.**

**Ans.**

The required query is:

SELECT DISTINCT W.FIRST\_NAME, T.WORKER\_TITLE

FROM Worker W

INNER JOIN Title T

ON W.WORKER\_ID = T.WORKER\_REF\_ID

AND T.WORKER\_TITLE in ('Manager');

Copy

**Q-25. Write an SQL query to fetch duplicate records having matching data in some fields of a table.**

**Ans.**

The required query is:

SELECT WORKER\_TITLE, AFFECTED\_FROM, COUNT(\*)

FROM Title

GROUP BY WORKER\_TITLE, AFFECTED\_FROM

HAVING COUNT(\*) > 1;

Copy

**Q-26. Write an SQL query to show only odd rows from a table.**

**Ans.**

The required query is:

SELECT \* FROM Worker WHERE MOD (WORKER\_ID, 2) <> 0;

Copy

**Q-27. Write an SQL query to show only even rows from a table.**

**Ans.**

The required query is:

SELECT \* FROM Worker WHERE MOD (WORKER\_ID, 2) = 0;

Copy

**Q-28. Write an SQL query to clone a new table from another table.**

**Ans.**

The general query to clone a table with data is:

SELECT \* INTO WorkerClone FROM Worker;

Copy

The general way to clone a table without information is:

SELECT \* INTO WorkerClone FROM Worker WHERE 1 = 0;

Copy

An alternate way to clone a table (for MySQL) without data is:

CREATE TABLE WorkerClone LIKE Worker;

Copy

**Q-29. Write an SQL query to fetch intersecting records of two tables.**

**Ans.**

The required query is:

(SELECT \* FROM Worker)

INTERSECT

(SELECT \* FROM WorkerClone);

Copy

**Q-30. Write an SQL query to show records from one table that another table does not have.**

**Ans.**

The required query is:

SELECT \* FROM Worker

MINUS

SELECT \* FROM Title;

Copy

**Q-31. Write an SQL query to show the current date and time.**

**Ans.**

The following MySQL query returns the current date:

SELECT CURDATE();

Copy

And the following MySQL query returns the current date and time:

SELECT NOW();

Copy

Here is a SQL Server query that returns the current date and time:

SELECT getdate();

Copy

Find this Oracle query that also returns the current date and time:

SELECT SYSDATE FROM DUAL;

Copy

**Q-32. Write an SQL query to show the top n (say 10) records of a table.**

**Ans.**

MySQL query to return the top n records using the LIMIT method:

SELECT \* FROM Worker ORDER BY Salary DESC LIMIT 10;

Copy

SQL Server query to return the top n records using the TOP command:

SELECT TOP 10 \* FROM Worker ORDER BY Salary DESC;

Copy

Oracle query to return the top n records with the help of ROWNUM:

SELECT \* FROM (SELECT \* FROM Worker ORDER BY Salary DESC)

WHERE ROWNUM <= 10;

Copy

**Q-33. Write an SQL query to determine the nth (say n=5) highest salary from a table.**

**Ans.**

MySQL query to find the nth highest salary:

SELECT Salary FROM Worker ORDER BY Salary DESC LIMIT n-1,1;

Copy

SQL Server query to find the nth highest salary:

SELECT TOP 1 Salary

FROM (

SELECT DISTINCT TOP n Salary

FROM Worker

ORDER BY Salary DESC

)

ORDER BY Salary ASC;

Copy

**Q-34. Write an SQL query to determine the 5th highest salary without using the TOP or limit method.**

**Ans.**

The following query is using the correlated subquery to return the 5th highest salary:

SELECT Salary

FROM Worker W1

WHERE 4 = (

SELECT COUNT( DISTINCT ( W2.Salary ) )

FROM Worker W2

WHERE W2.Salary >= W1.Salary

);

Copy

Use the following generic method to find the nth highest salary without using TOP or limit.

SELECT Salary

FROM Worker W1

WHERE n-1 = (

SELECT COUNT( DISTINCT ( W2.Salary ) )

FROM Worker W2

WHERE W2.Salary >= W1.Salary

);

Copy

**Q-35. Write an SQL query to fetch the list of employees with the same salary.**

**Ans.**

The required query is:

Select distinct W.WORKER\_ID, W.FIRST\_NAME, W.Salary

from Worker W, Worker W1

where W.Salary = W1.Salary

and W.WORKER\_ID != W1.WORKER\_ID;

Copy

**Q-36. Write an SQL query to show the second-highest salary from a table.**

**Ans.**

The required query is:

Select max(Salary) from Worker

where Salary not in (Select max(Salary) from Worker);

Copy

**Q-37. Write an SQL query to show one row twice in the results from a table.**

**Ans.**

The required query is:

select FIRST\_NAME, DEPARTMENT from worker W where W.DEPARTMENT='HR'

union all

select FIRST\_NAME, DEPARTMENT from Worker W1 where W1.DEPARTMENT='HR';

Copy

**Q-38. Write an SQL query to fetch intersecting records of two tables.**

**Ans.**

The required query is:

(SELECT \* FROM Worker)

INTERSECT

(SELECT \* FROM WorkerClone);

Copy

**Q-39. Write an SQL query to fetch the first 50% of records from a table.**

**Ans.**

The required query is:

SELECT \*

FROM WORKER

WHERE WORKER\_ID <= (SELECT count(WORKER\_ID)/2 from Worker);

Copy

**Q-40. Write an SQL query to fetch the departments that have less than five people in them.**

**Ans.**

The required query is:

SELECT DEPARTMENT, COUNT(WORKER\_ID) as 'Number of Workers' FROM Worker GROUP BY DEPARTMENT HAVING COUNT(WORKER\_ID) < 5;

Copy

**Q-41. Write an SQL query to show all departments along with the number of people in there.**

**Ans.**

The following query returns the expected result:

SELECT DEPARTMENT, COUNT(DEPARTMENT) as 'Number of Workers' FROM Worker GROUP BY DEPARTMENT;

Copy

**Q-42. Write an SQL query to show the last record from a table.**

**Ans.**

The following query will return the last record from the Worker table:

Select \* from Worker where WORKER\_ID = (SELECT max(WORKER\_ID) from Worker);

Copy

**Q-43. Write an SQL query to fetch the first row of a table.**

**Ans.**

The required query is:

Select \* from Worker where WORKER\_ID = (SELECT min(WORKER\_ID) from Worker);

Copy

**Q-44. Write an SQL query to fetch the last five records from a table.**

**Ans.**

The required query is:

SELECT \* FROM Worker WHERE WORKER\_ID <=5

UNION

SELECT \* FROM (SELECT \* FROM Worker W order by W.WORKER\_ID DESC) AS W1 WHERE W1.WORKER\_ID <=5;

Copy

**Q-45. Write an SQL query to print the name of employees having the highest salary in each department.**

**Ans.**

The required query is:

SELECT t.DEPARTMENT,t.FIRST\_NAME,t.Salary from(SELECT max(Salary) as TotalSalary,DEPARTMENT from Worker group by DEPARTMENT) as TempNew

Inner Join Worker t on TempNew.DEPARTMENT=t.DEPARTMENT

and TempNew.TotalSalary=t.Salary;

Copy

**Q-46. Write an SQL query to fetch three max salaries from a table.**

**Ans.**

The required query is:

SELECT distinct Salary from worker a WHERE 3 >= (SELECT count(distinct Salary) from worker b WHERE a.Salary <= b.Salary) order by a.Salary desc;

Copy

**Q-47. Write an SQL query to fetch three min salaries from a table.**

**Ans.**

The required query is:

SELECT distinct Salary from worker a WHERE 3 >= (SELECT count(distinct Salary) from worker b WHERE a.Salary >= b.Salary) order by a.Salary desc;

Copy

**Q-48. Write an SQL query to fetch nth max salaries from a table.**

**Ans.**

The required query is:

SELECT distinct Salary from worker a WHERE n >= (SELECT count(distinct Salary) from worker b WHERE a.Salary <= b.Salary) order by a.Salary desc;

Copy

**Q-49. Write an SQL query to fetch departments along with the total salaries paid for each of them.**

**Ans.**

The required query is:

 SELECT DEPARTMENT, sum(Salary) from worker group by DEPARTMENT;

Copy

**Q-50. Write an SQL query to fetch the names of workers who earn the highest salary.**

**Ans.**

The required query is:

SELECT FIRST\_NAME, SALARY from Worker WHERE SALARY=(SELECT max(SALARY) from Worker);

Copy

**Summary – Master SQL Queries with 50 Practice Questions and Answers.**

We hope you enjoyed solving the SQL exercises and learned something new along the way.

Stay tuned for our next post, where we’ll bring you even more challenging SQL query questions to sharpen your proficiency.

Hey there, don’t hit that exit button just yet! We’ve got a treasure trove of resourceful tutorials that’ll leave you hungry for more.

* [SQL Exercises – Complex Queries](https://www.techbeamers.com/sql-exercises/)
* [Top SQL Interview Questions One Should Know Beforehand](https://www.techbeamers.com/sql-interview-questions-answers-experienced/)
* [25 SQL Performance Interview Questions and Answers](https://www.techbeamers.com/sql-performance-interview-questions-answers/)
* [Top 30 PL SQL Interview Questions and Answers for Freshers](https://www.techbeamers.com/pl-sql-interview-questions-answers-freshers/)

If you found this helpful, please consider sharing it with others and following us on our social media platforms  ([Facebook](https://www.facebook.com/TechBeamersWorld/)/[Twitter](https://twitter.com/TechBeamers)) for more helpful resources.

-Thank you for choosing TechBeamers.

## SQL Exercises

The section below outlines the SQL queries to create test tables and demo data. Execute these commands to proceed.

### Sample Tables and Insert Demo Data

#### Step-1 (Create Table)

It is better to create the test data in a dedicated and separate database. Hence, let’s first create a database for our testing purpose.

CREATE database SQLTest;

USE SQLTest;

Copy

Here, you’ll be creating two tables, namely EMPLOYEE and DEPARTMENT. The exercises will revolve around them.

CREATE TABLE DEPARTMENT

(

DEPTCODE INT(10),

DeptName CHAR(30),

LOCATION VARCHAR(33)

);

CREATE TABLE EMPLOYEE

(

EmpCode INT(4),

EmpFName VARCHAR(15),

EmpLName VARCHAR(15),

Job VARCHAR(45),

Manager CHAR(4),

HireDate DATE,

Salary INT(6),

Commission INT(6),

DEPTCODE INT(2)

);

Copy

#### Step-2 (Alter Table)

We have now created the desired SQL tables. Next, you should run the below commands to change the table structure. It is sometimes quite useful that you know how to alter existing table properties.

ALTER TABLE DEPARTMENT

ADD PRIMARY KEY (DEPTCODE);

ALTER TABLE DEPARTMENT

CHANGE COLUMN DEPTCODE DEPTCODE INT(10) NOT NULL;

ALTER TABLE DEPARTMENT

CHANGE COLUMN DeptName DeptName CHAR(30) UNIQUE;

ALTER TABLE DEPARTMENT

CHANGE COLUMN LOCATION LOCATION VARCHAR(33) NOT NULL;

ALTER TABLE DEPARTMENT

CHANGE COLUMN DeptName DeptName VARCHAR(15) UNIQUE;

ALTER TABLE EMPLOYEE

ADD PRIMARY KEY (EmpCode);

ALTER TABLE EMPLOYEE

CHANGE COLUMN EmpCode EmpCode INT(4) NOT NULL;

ALTER TABLE EMPLOYEE

ADD FOREIGN KEY (DEPTCODE)

REFERENCES DEPARTMENT(DEPTCODE);

ALTER TABLE EMPLOYEE

CHANGE COLUMN Salary Salary DECIMAL(6,2);

ALTER TABLE EMPLOYEE

ADD COLUMN DOB DATE

AFTER EmpLName;

ALTER TABLE EMPLOYEE

DROP COLUMN DOB;

Copy

#### Step-3 (Populate Table)

Below INSERT statement will fill the above tables with demo data you can use to run queries.

INSERT INTO DEPARTMENT VALUES (10, 'FINANCE', 'EDINBURGH'),

(20,'SOFTWARE','PADDINGTON'),

(30, 'SALES', 'MAIDSTONE'),

(40,'MARKETING', 'DARLINGTON'),

(50,'ADMIN', 'BIRMINGHAM');

INSERT INTO EMPLOYEE

VALUES (9369, 'TONY', 'STARK', 'SOFTWARE ENGINEER', 7902, '1980-12-17', 2800,0,20),

(9499, 'TIM', 'ADOLF', 'SALESMAN', 7698, '1981-02-20', 1600, 300,30),

(9566, 'KIM', 'JARVIS', 'MANAGER', 7839, '1981-04-02', 3570,0,20),

(9654, 'SAM', 'MILES', 'SALESMAN', 7698, '1981-09-28', 1250, 1400, 30),

(9782, 'KEVIN', 'HILL', 'MANAGER', 7839, '1981-06-09', 2940,0,10),

(9788, 'CONNIE', 'SMITH', 'ANALYST', 7566, '1982-12-09', 3000,0,20),

(9839, 'ALFRED', 'KINSLEY', 'PRESIDENT', 7566, '1981-11-17', 5000,0, 10),

(9844, 'PAUL', 'TIMOTHY', 'SALESMAN', 7698, '1981-09-08', 1500,0,30),

(9876, 'JOHN', 'ASGHAR', 'SOFTWARE ENGINEER', 7788, '1983-01-12',3100,0,20),

(9900, 'ROSE', 'SUMMERS', 'TECHNICAL LEAD', 7698, '1981-12-03', 2950,0, 20),

(9902, 'ANDREW', 'FAULKNER', 'ANAYLYST', 7566, '1981-12-03', 3000,0, 10),

(9934, 'KAREN', 'MATTHEWS', 'SOFTWARE ENGINEER', 7782, '1982-01-23', 3300,0,20),

(9591, 'WENDY', 'SHAWN', 'SALESMAN', 7698, '1981-02-22', 500,0,30),

(9698, 'BELLA', 'SWAN', 'MANAGER', 7839, '1981-05-01', 3420, 0,30),

(9777, 'MADII', 'HIMBURY', 'ANALYST', 7839, '1981-05-01', 2000, 200, NULL),

(9860, 'ATHENA', 'WILSON', 'ANALYST', 7839, '1992-06-21', 7000, 100, 50),

(9861, 'JENNIFER', 'HUETTE', 'ANALYST', 7839, '1996-07-01', 5000, 100, 50);

Copy

### SQL Exercises for Basic to Advanced Queries

#1 Create a query that displays EMPFNAME, EMPLNAME, DEPTCODE, DEPTNAME, LOCATION from EMPLOYEE, and DEPARTMENT tables. Make sure the results are in the ascending order based on the EMPFNAME and LOCATION of the department.

SELECT E.EMPFNAME, E.EMPLNAME, E.DEPTCODE,

D.DEPTNAME, D.LOCATION

FROM EMPLOYEE E, DEPARTMENT D

WHERE E.DEPTCODE = D.DEPTCODE

ORDER BY E.EMPFNAME, D.LOCATION;

Copy

#2 Display EMPFNAME and “TOTAL SALARY” for each employee

SELECT EMPFNAME, SUM(COMMISSION+SALARY) AS "TOTAL SALARY" FROM EMPLOYEE GROUP BY EMPCODE;

Copy

#3 Display MAX and 2nd MAX SALARY from the EMPLOYEE table.

SELECT

(SELECT MAX(SALARY) FROM EMPLOYEE) MAXSALARY,

(SELECT MAX(SALARY) FROM EMPLOYEE

WHERE SALARY NOT IN (SELECT MAX(SALARY) FROM EMPLOYEE )) as 2ND\_MAX\_SALARY;

Copy

#4 Display the TOTAL SALARY drawn by an analyst working in dept no 20

SELECT SUM(SALARY+COMMISSION) AS TOTALSALARY FROM EMPLOYEE

WHERE JOB = 'ANALYST' AND DEPTCODE = 20;

Copy

#5 Compute average, minimum and maximum salaries of the group of employees having the job of ANALYST.

SELECT AVG(Salary) AS AVG\_SALARY, MIN(Salary) AS MINSALARY, MAX(Salary) AS MAXSALARY

FROM EMPLOYEE WHERE Job = 'ANALYST';

Copy

Please note that we’ll be adding more and more [SQL queries](https://www.techbeamers.com/sql-query-questions-answers-for-practice/) to this post based on your feedback. So, please do share your questions with us.