**Introduction to SQL with Activity Sheet**

**What is SQL?**

Structured Query Language, SQL for short, is used to extract information from a database, create or change tables, and store data in a structured way. Most relational database management systems use SQL as the standard language to access and modify data. SQL was originally developed at IBM during the 1970’s and is still in wide use today. It became an ANSI standard and ISO standard around 1986.

The first step in working with SQL is creating a table of data in your database of choice. Each table individually is like a single Excel sheet with rows and columns. A relational database is like a collection of Excel sheets with some tables related to other tables via a shared column. With SQL the user can build their database by creating tables, altering tables with new columns, or completely deleting tables by dropping them. Users have a few options for entering in data by importing it depending on the software or they can do so manually by creating, deleting, or updating data. The SQL commands for those operations are CREATE, DELETE, DROP, UPDATE, and INSERT.

For extracting data, SQL offers a flexible framework to request data from a database using the keywords SELECT, FROM, WHERE, and JOIN. Other programs leverage SQL to access databases via embedded SQL, SQL modules, and call-level interface (CLI). With these keywords, users create queries to request access to data stored in tables.

**Why use SQL?**

SQL is good for ad hoc queries where the user’s analysis depends on summarizing a subset of data. Data manipulation is one of SQL’s main advantages including updating tables, joining data from different sources, and filtering out unneeded data. SQL can carry out multiple transactions in one query and maintain quick performance on larger datasets. Data storage is another main advantage since using relational databases with SQL ensures constant availability and persistence of data.

Some limitations of SQL are that it takes a while to set-up tables and it is usually implemented on one machine so horizonal scaling is difficult. Since SQL is completely tabular based it is more difficult to implement complex logic or object-oriented applications. Sometimes users have difficulty using SQL if they want to control how an operation is carried out whereas SQL is declarative so only predetermined operations can be carried out making it inflexible for some problems. [7]

**Tools that use SQL.**

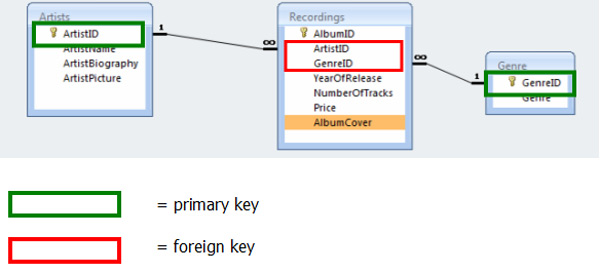
SQL is a programming language used with relational database management systems such as Oracle, Microsoft SQL Server, and Access. In order to use SQL two things are needed, a server and an interface. The server is where the data is stored and the interface is how the data is accessed. There are many options but for an introduction to the top 5 see (<https://learn.g2.com/what-is-sql-server>).

**What are relational databases?**

A relation database is defined as “a set of formally described tables from which data can be accessed or reassembled” [3]. It was designed as a move away from storing data via hierarchical structures and towards storing data by record in table format. Relational tables are similar to Excel in that data is stored in rows and columns but different in that tables must contain a column of unique values called a primary key that identifies the row or record. The word “relational” describes how tables are linked to each other using foreign keys to associate one row with the corresponding primary key in another table.

Taking a closer look at the primary and foreign key concept, here is a short example. In the picture below there are 3 tables each with a primary key (ArtistID, AlbumID, and GenreID) and 1 table with 2 foreign keys (ArtistID and GenreID). The foreign keys in the Recordings table point to the primary keys in the Artists and Genre tables which link these tables together.

Let’s say we are interested in the album Abbey Road and want more information on the artist and genre. From the Recordings table we could get information on the artist ID, genre ID, release year, number of songs, price, and album cover. Next, we would use the foreign key ArtistID to find the correct row in the Artists table where we could get information on the artist name, biography, and artist picture. Back to the Recordings table, we would use the other foreign key GenreID to access the proper row in the Genre table giving us more information on the genre of the album. Foreign keys and primary keys are how you can utilize multiple tables and find connections in your data.

<https://www.teach-ict.com/as_a2_ict_new/ocr/AS_G061/315_database_concepts/terminology/miniweb/pg13.htm>

**Entering Data: Create, Drop, Alter Tables**

For this example, we will first create a new table with employee data from a fictional company then we will change some columns in the table to correct mistakes and add on a new column. Finally, we will drop the table which deletes everything in it.

**Create:**

In order to create a table in SQL, the first step is to create a schema for it. A schema lets the program know what kind of data to expect so it can create the correct containers to hold the data. SQL must know ahead of time what data types to expect which helps users protect against the wrong data going into the table. The CREATE statement is how a schema is built in SQL and is the first part in manually entering data.

Every create statement must include the create keyword, name of the table, column names, and column datatypes. The keyword “Create table” is followed by a table name that starts with an alphabetic character or an underscore and only contains alphanumeric characters. The database settings control case significance so upper and lower cases may make a difference. Every column needs to be listed within the parenthesis along with its corresponding datatype.

There are several datatypes including numeric, date/time, character/string, Unicode, binary, and other. Each category has subtypes, but the most used ones are int and float for numeric, datetime for dates, and char or varchar for strings. The specified datatype is important because it determines what kind of input is acceptable for each column. For example, if a column’s datatype is INT then it cannot accept words as input or else an error would result. The datatype limits the type of data that can be stored in that column. Additional constraints can be added such as unique, not null, and primary/foreign key specifications. For more on this topic, visit <https://www.w3schools.com/sql/sql_constraints.asp>.

Data Type Table with Limits

|  |  |  |
| --- | --- | --- |
| Numeric | Lowest | Highest |
| Bit | 0 | 1 |
| Tinyint | 0 | 255 |
| Smallint | -32,768 | 32,768 |
| Int | -2,147,483,648 | 2,147,483,647 |
| Bigint | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,808 |
| Decimal | -10^38 | 10^38 |
| Numeric | -10^38 | 10^38 |
| Float | -1.79E +308 | 1.79E +308 |
| Real | -3.4E + 38 | 3.4E + 38 |

Date Data Types

|  |  |
| --- | --- |
| Date and Time | Format |
| Date | YYYY-MM-DD |
| Time | HH:MI:SS |
| Datetime | YYYY-MM-DD HH:MI:SS |
| Timestamp | Numbers of second since Unix epoch |
| Year | YY or YYYY format |

Character Data Types

|  |  |
| --- | --- |
| Character / String | Description |
| Char | Fixed length with max length of 8,000 characters |
| Varchar | Variable length with max length of 8,000 characters |
| Varchar(Max) | Variable length with specified max length |
| Text | Variable length with max size of 2GB data |

<https://www.journaldev.com/16774/sql-data-types>

**General syntax:**

CREATE TABLE table\_name (

Column\_name\_1 data\_type,

Column\_name\_2 data\_type,

Column\_name\_3 data\_type

);

**Code Example:**

CREATE TABLE Employee (

WorkerID smallint UNIQUE,

Name varchar(50),

Department varchar(100),

Salary int,

JobTitle varchar(255)

);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
|  |  |  |  |  |

In order to create a copy of an already existing table use the abbreviated version below. Specify the new table name and the columns and table to copy but do not include any datatypes since they are already set.

**General Syntax:**

CREATE TABLE new\_table AS

SELECT column\_one, column\_two

FROM old\_table

**Code Example:**

CREATE TABLE employee\_copy AS

SELECT WorkerID, Name, Department

FROM employee

|  |  |  |
| --- | --- | --- |
| WorkerID | Name | Department |
|  |  |  |

**Alter:**

The Alter statement is used for changing columns in a table after it was created and does not refer to changing the data. The options are to add, delete, or change the datatype or to change the constraints of a column in a table.

**General syntax:**

**Adds a column**

ALTER TABLE existing\_table

ADD new\_column datatype;

**Drops a column**

ALTER TABLE existing\_table

DROP COLUMN column\_name;

**Change the datatype of a column**

ALTER TABLE existing\_table

ALTER COLUMN column\_name datatype;

**Change the datatype of a column in MySQL or Oracle**

ALTER TABLE existing\_table

MODIFY COLUMN column\_name datatype;

**Code Example:**

ALTER TABLE employee

ADD StartDate datetime;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle | StartDate |
|  |  |  |  |  |  |

ALTER TABLE employee

DROP COLUMN StartDate;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
|  |  |  |  |  |

ALTER TABLE employee

ALTER COLUMN Salary bigint;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
|  |  |  |  |  |

**Insert Into:**

Use the INSERT INTO statement to add data to an already created dataset. Many database tools have options where you can import data from other sources such as Excel, csv, and txt files. This allows users to easily import data and specify column and table properties so be sure to check for a Data Import Wizard before inserting data manually. This method of inserting rows is good for smaller tables or adding new rows to an existing table. If you are adding values to all columns, then you don’t need to specify the columns but if you are only adding data to a few then the columns need to be specified. For character or string values, enclose them in single or double quotes.

**General Syntax:**

**Insert data into specific columns**

INSERT INTO table\_name (column\_one, column\_two)

VALUES (value\_one, value\_two)

**Specify values for all columns**

INSERT INTO table\_name

VALUES (value\_one, value\_two, value\_three, …)

**Insert multiple rows**

INSERT INTO table\_name (column\_one, column\_two)

VALUES

(value\_A\_1, value\_B\_1),

(value\_A\_2, value\_B\_2),

(value\_A\_3, value\_B\_3)

**Code Example:**

INSERT INTO employees

VALUES

(1, ‘Jane’, ‘Sales’, 50000, ‘Sales Associate’),

(2, ‘Max’, ‘HR’, 45000, ‘Benefits Specialist’)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |

INSERT INTO employees (WorkerId, Name, Department, Salary, JobTitle)

VALUES (3, ‘Lee’, ‘R & D’, 74000, ‘Researcher’)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |
| 3 | Lee | R & D | 74,000 | Researcher |

**Drop and Truncate:**

If a table needs to be deleted completely, the drop statement removes it and deletes all the data from the table. Be careful with dropping tables since this operation is irreversible after the command is executed and committed. This statement deletes the schema, data, and table from the database.

If you want to delete the data in a table but keep its schema, which includes the columns and datatype specifications, then use the truncate statement. This will not delete the table only the rows and it will return an empty table.

**General Syntax:**

DROP TABLE table\_name;

TRUNCATE TABLE table\_name;

**Code Example:**

DROP TABLE employees;

< No table>

TRUNCATE TABLE employees;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
|  |  |  |  |  |

**Basic query**

The keywords SELECT, FROM, and WHERE form the basis of any query and can be used to extract any subset of data from a table. We will look at them one by one then combine them to form our first query. In this case the input will be the Employees table and the output will be a filtered subset of that table.

Employees table as created earlier:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |
| 3 | Lee | R & D | 74,000 | Researcher |
| 4 | Sam | HR | 80,000 | Manager |
| 5 | Haley | Finance | 60,000 | Accountant |

**Select:** The SELECT keyword chooses the columns from the table to be included in the final output. Any selection of columns can be chosen by listing the desired column names separated by a comma but at least one column must be listed. The asterisk \* is used to signify all columns should be included instead of typing them out individually.

**From:** The FROM keyword specifies which table to extract data from. In this example there is only one table but in databases with multiple tables SQL needs to be directed to a specific table to query.

**General Syntax:**

SELECT \*

FROM table\_name;

**Code examples:**

SELECT Name, Salary, JobTitle

FROM employees

|  |  |  |
| --- | --- | --- |
| Name | Salary | JobTitle |
| Jane | 50,000 | Sales Associate |
| Max | 45,000 | Benefits Specialist |
| Lee | 74,000 | Researcher |
| Sam | 80,000 | Manager |
| Haley | 60,000 | Accountant |

SELECT \*

FROM employees

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |
| 3 | Lee | R & D | 74,000 | Researcher |
| 4 | Sam | HR | 80,000 | Manager |
| 5 | Haley | Finance | 60,000 | Accountant |

Notice how the resulting table only returns the columns specified and the \* returns the original table. In both cases, all rows are returned as that is the default behavior.

**Where:** The WHERE keyword selects which rows to include by setting a criteria that each row must meet in order to be included in the output. The criteria can be set using equality constraints (=), greater or less than conditions (> or <), and lists (IN, NOT IN) to determine which rows to return. The WHERE clause contains three parts: the column, the comparison operator, and the threshold or desired value. A query does not require a where clause but if one is used then at least one column needs to be constrained but multiple columns can be specified and joined using AND or OR between them.

**General Syntax:**

SELECT \*

FROM table\_name

WHERE condition;

**Code examples:**

SELECT \*

FROM employees

WHERE department = ‘HR’

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 2 | Max | HR | 45,000 | Benefits Specialist |
| 4 | Sam | HR | 80,000 | Manager |

Since the Department column contains characters instead of numeric values, the condition uses an equality constraint with HR in quotes.

SELECT \*

FROM employees

WHERE salary < 55000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |

SELECT \*

FROM employees

WHERE name in (‘Sam’, ‘Haley’)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 4 | Sam | HR | 80,000 | Manager |
| 5 | Haley | Finance | 60,000 | Accountant |

SELECT \*

FROM employees

WHERE department NOT IN (‘Sales’, ‘R & D’, ‘Finance’) AND salary > 60000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 4 | Sam | HR | 80,000 | Manager |

**Aggregations**

Aggregations are a powerful way to analyze different groups within your data. SQL allows you to analyze groups by summarizing numerical columns using counts, sums, and averages. The GROUP BY key word gives you control over how to form groups while the HAVING keyword allows you to set the criteria for including a group in the final output.

**Group By**

Let us start by exploring what groups mean in SQL. Each column can be condensed to a list of unique values by removing duplicates. If only one column was selected in the GROUP BY statement, then each unique value would be its own group.

Take for example the employee table and specifically the Department column. Since there are 4 unique values there will be 4 groups including Sales, HR, R & D, and Finance. The HR group will have 2 rows belonging to it, the second and fourth row, while all other groups will only have 1 row in the group. We can only group based on unique values so grouping Salary by bin (0-25,000, 25,000- 50,000, etc) would not be possible.

Employee table as created originally:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |
| 3 | Lee | R & D | 74,000 | Researcher |
| 4 | Sam | HR | 80,000 | Manager |
| 5 | Haley | Finance | 60,000 | Accountant |

**General Syntax:**

SELECT column\_name, aggregate\_function(column\_name)

FROM table\_name

GROUP BY column\_name

**Code example:**

SELECT department

FROM employees

GROUP BY department

|  |
| --- |
| Department |
| Sales |
| HR |
| R & D |
| Finance |

SQL has the flexibility to make different groups by combining columns to make combinations. If multiple columns are selected, then each combination of the unique values from each column make up a group.

Take the revenue table shown below for example, in order to form groups based on unique dates then both the Year and Quarter column are used in the group by statement. The groups would be 2016 Q1, 2016 Q2, 2016 Q3…, 2017 Q4. However, this grouping would not be useful with this data set since there is only one row per group.

Sales table example:

|  |  |  |
| --- | --- | --- |
| Year | Quarter | Revenue (M) |
| 2016 | 1 | 1.1 |
| 2016 | 2 | 1.3 |
| 2016 | 3 | 2.5 |
| 2016 | 4 | 2.9 |
| 2017 | 1 | 3.1 |
| 2017 | 2 | 2.8 |
| 2017 | 3 | 2.6 |

**Code example:**

SELECT year, quarter

FROM revenue

GROUP BY year, quarter

|  |  |
| --- | --- |
| Year | Quarter |
| 2016 | 1 |
| 2016 | 2 |
| 2016 | 3 |
| 2016 | 4 |
| 2017 | 1 |
| 2017 | 2 |
| 2017 | 3 |

GROUP BY statements are used mainly to perform calculations to summarize groups by the aggregate functions: COUNT, MAX, MIN, SUM, and AVG. By specifying one of these functions in the SELECT statement, an additional column is returned summarizing the group by that function. This could be used for a range of calculations such as finding the highest salary in each group, the number of rows per group, or the group mean. These functions can only apply to numeric columns.

The syntax is to specify the group columns first in the SELECT statement then add the aggregate function with the column it applies to in the parentheses.

Employee table as originally created:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WorkerID | Name | Department | Salary | JobTitle |
| 1 | Jane | Sales | 50,000 | Sales Associate |
| 2 | Max | HR | 45,000 | Benefits Specialist |
| 3 | Lee | R & D | 74,000 | Researcher |
| 4 | Sam | HR | 80,000 | Manager |
| 5 | Haley | Finance | 60,000 | Accountant |

**Code example:**

SELECT department, max(salary)

FROM employees

GROUP BY department

|  |  |
| --- | --- |
| Department | Salary |
| Sales | 50,000 |
| R & D | 74,000 |
| HR | 80,000 |
| Finance | 60,000 |

**Having**

Having is similar to the WHERE clause in that it selects rows to include in the final output. However, HAVING is used for groups since WHERE does not apply for aggregate functions. The condition in the HAVING clause needs to reflect the aggregate function in the SELECT statement otherwise it does not work.

**General Syntax:**

SELECT column\_name, aggregate\_function(column\_name)

FROM table\_name

GROUP BY column\_name

HAVING aggregate\_function(column\_name) (>,=,<, IN, NOT IN) value

**Code example:**

SELECT department, sum(salary)

FROM employees

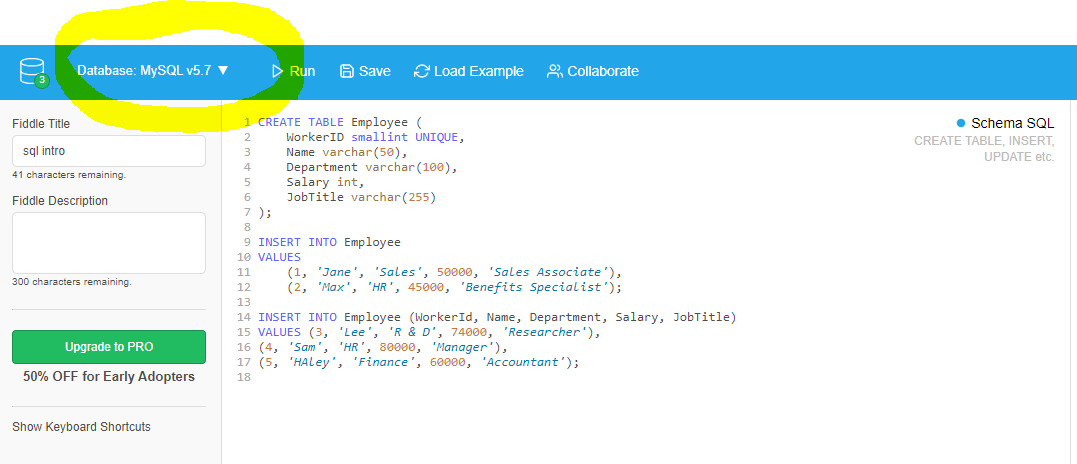
GROUP BY department

HAVING sum(salary) > 60000

|  |  |
| --- | --- |
| Department | Salary |
| R & D | 74,000 |
| HR | 125,000 |

**SQL Practice**

Go to db-fiddle.com to practice SQL using the examples from this guide and try making your own tables. Before you start coding, set which version of SQL you would like to use in the upper left corner.



Give your page a title in the Fiddle Title box and start coding in the Schema SQL section. To create a new table, start with a CREATE statement followed by INSERT statements to add data. The code will be color coded according to the following guidelines: dark blue for key words, black for names created by the user, orange for numbers, light blue for strings, and brown for data types. If you enter a string like ‘Jane’ and it is not colored light blue then try re-typing the single quotes since it can be copied incorrectly. Do not forget the semicolon (;) between each statement. Once the code looks correct try to run it by clicking the run button on the top bar.

Code to create the employee table:

CREATE TABLE Employee (

WorkerID smallint UNIQUE,

Name varchar(50),

Department varchar(100),

Salary int,

JobTitle varchar(100)

);

INSERT INTO Employee

VALUES

(1, 'Jane', 'Sales', 50000, 'Sales Associate'),

(2, 'Max', 'HR', 45000, 'Benefits Specialist'),

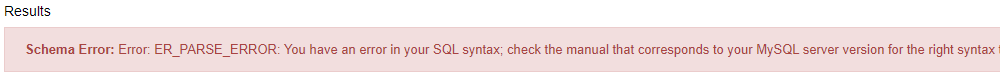
(3, 'Lee', 'R & D', 74000, 'Researcher'),

(4, 'Sam', 'HR', 80000, 'Manager'),

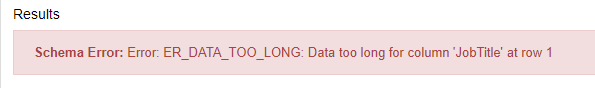
(5, 'Haley', 'Finance', 60000, 'Accountant');



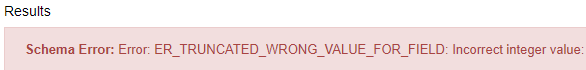
If you got the following error, go back and check for any syntax errors in the code. This can often be solved by correcting a typo in the area specified by the error.



This error occurs when the schema specified in the CREATE statement is not set correctly. The data has entries longer than specified in the varchar(10) portion so the parameter needs to be greater than the longest entry in order to allocate enough space. To correct this error, change the parameter to a larger number like varchar(100).



Specifying the correct data type in the schema is important since it allows SQL to plan out how to build the table. If you specify one data type and try to insert another data type, you will get the following error. This commonly occurs if the data is a string or character and the schema is expecting a numeric data type like int or double. SQL can’t handle strings in an int column but SQL will allow numeric values in a varchar() column.



Once the Schema SQL code is running without errors, go over to the right side in the Query SQL section and enter the following code.

Query SQL Code:

SELECT \*

FROM Employee

If there are no errors, a table should appear in the results section with the column names specified in your CREATE statement and with the data from the INSERT statements.



Now that the basic query is working, play around with selecting only certain columns and rows.

Examples to try:

1. Output only the WorkerID and Department column from the employee table:

Select WorkerID, Department

From Employee

1. Select rows where salary is greater than 50,000:

Select \*

From Employee

Where salary > 50000

1. Try to form groups based on JobTitle and get the minimum salary:

Select JobTitle, min(salary)

From employee

Group by JobTitle

**Exercise:**

Pretend you are a business manager for a home goods store and would like to analyze your sales over the past year. Each quarter you recorded the total sales in thousands from various products sold in different regions in a table called Sales. Now you would like to identify areas with strong sales and areas that need some work. To start with this example, go to db-fiddle.com and copy in the starter code below.

1. Fill in the necessary data types based on the data from the INSERT statement. Remember to specify numeric columns appropriately and for the text data specify the maximum for the column length.

CREATE TABLE Sales (

ProductID smallint UNIQUE,

ProductName \_\_\_\_\_\_\_\_,

ProductGroup varchar(100),

Region varchar(20),

Quarter \_\_\_\_\_\_\_,

Year int,

TotalSales \_\_\_\_\_\_\_\_

);

INSERT INTO Sales

VALUES (1, 'Toaster', 'Appliances', 'North', 1, 2019, 1150.65),

(2, 'Toaster', 'Appliances', 'South', 1, 2019, 2658.79),

(3, 'Tea Cup', 'Kitchen', 'East', 1, 2019, 1290.00),

(4, 'Coffee Maker', 'Appliances', 'North', 1, 2019, 76895.95),

(5, 'Blender', 'Appliances', 'East', 1, 2019, 2456.95),

(6, 'Tea Set', 'Kitchen', 'North', 2, 2019, 105.00),

(7, 'Silverware', 'Kitchen', 'West', 2, 2019, 3460.90),

(8, 'Pan', 'Kitchen', 'West', 2, 2019, 7649.50),

(9, 'Decorative sign', 'Decor', 'South', 2, 2019, 280.90),

(10, 'Towels', 'Bath', 'South', 2, 2019, 7566.50);

1. Once the CREATE and INSERT statement run without any errors, go to the Query SQL box and enter in the correct SELECT statement to display the data table.
2. Select only the following columns: ProductName, ProductGroup, and Region.
3. Select all the columns but only rows with East as the Region.
4. Select all the columns but only rows with Quarter 2.
5. Set up a group based on the ProductGroup that returns the average TotalSales.
6. Set up a group based on the Quarter and Year that returns the sum of sales for each group.
7. Return a list of Regions where the sum of sales was greater than 10,000.
8. Using only data from Quarter 2, return the count of items for each ProductGroup.
9. From the Appliances, get minimum TotalSales value.

**Answers:**

1.

CREATE TABLE Sales (

ProductID smallint UNIQUE,

ProductName varchar(50),

ProductGroup varchar(100),

Region varchar(20),

Quarter smallint,

Year int,

TotalSales double

);

2.

select \*

from Sales

3.

select ProductName, ProductGroup, Region

from Sales

4.

select \*

from Sales

where Region = 'East'

5.

select \*

from Sales

where Quarter = 2

6.

select ProductGroup, avg(TotalSales)

from Sales

group by ProductGroup

7.

select Quarter, Year, sum(TotalSales)

from Sales

group by Quarter, Year

8.

select Region, sum(TotalSales)

from Sales

group by Region

having sum(TotalSales) > 10000

9.

select ProductGroup, count(\*)

from Sales

where quarter = 2

group by ProductGroup

10.

select ProductGroup, min(TotalSales)

from Sales

where ProductGroup = 'Appliances'

group by ProductGroup

**Links**

1. <https://www.db-fiddle.com/>
2. <https://docs.microsoft.com/en-us/sql/odbc/reference/structured-query-language-sql?redirectedfrom=MSDN&view=sql-server-ver15>
3. <https://www.edrm.net/glossary/relational-database/>
4. <https://capgemini.github.io/design/sql-vs-nosql/>
5. <https://searchdatamanagement.techtarget.com/definition/relational-database>
6. <https://www.oracle.com/database/what-is-a-relational-database/>
7. <https://www.loginradius.com/engineering/relational-database-management-system-rdbms-vs-nosql/>