Basic SQL

Here is a list of basic SQL commands (sometimes called clauses) used frequently.

SELECT and FROM

The SELECT part of a query determines which columns of the data to show in the results. There are also options you can apply to show data that is not a table column.

The example below shows three columns SELECTed FROM the “student” table and one calculated column. The database stores the studentID, FirstName, and LastName of the student. We can combine the First and the Last name columns to create the FullName calculated column.

```
SELECT studentID, FirstName, LastName, FirstName + ' ' +
LastName AS FullName
FROM student;
```

```
-----------+-------------------+------------+------------------------+
| studentID | FirstName         | LastName   | FullName               |
+-----------+-------------------+------------+------------------------+
|         1 | Monique           | Davis      | Monique Davis          |
|         2 | Teri              | Gutierrez  | Teri Gutierrez         |
|         3 | Spencer           | Pautier    | Spencer Pautier        |
|         4 | Louis             | Ramsey     | Louis Ramsey           |
|         5 | Alvin             | Greene     | Alvin Greene           |
|         6 | Sophie            | Freeman    | Sophie Freeman         |
|         7 | Edgar Frank "Ted" | Codd       | Edgar Frank "Ted" Codd |
|         8 | Donald D.         | Chamberlin | Donald D. Chamberlin   |
|         9 | Raymond F.        | Boyce      | Raymond F. Boyce       |
+-----------+-------------------+------------+------------------------+
9 rows in set (0.00 sec)
```

CREATE TABLE

`CREATE TABLE` does just what it sounds like: it creates a table in the database. You can specify the name of the table and the columns that should be in the table.

```
CREATE TABLE table_name (
    column_1 datatype,
    column_2 datatype,
    column_3 datatype
);
```

ALTER TABLE

`ALTER TABLE` changes the structure of a table. Here is how you would add a column to a database:

```
ALTER TABLE table_name
ADD column_name datatype;
```

CHECK

The CHECK constraint is used to limit the value range that can be placed in a column.
If you define a CHECK constraint on a single column it allows only certain values for this column. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row

The following SQL creates a CHECK constraint on the “Age” column when the “Persons” table is created. The CHECK constraint ensures that you can not have any person below 18 years.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18)
);
```

To allow naming of a CHECK constraint, and for defining
a CHECK constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

WHERE

(AND, OR, IN, BETWEEN, and LIKE)

The WHERE clause is used to limit the number of rows returned.
As an example, first we will show you a SELECT statement and
results *without* a WHERE statement. Then we will add a WHERE statement
that uses all five qualifiers above.

```
SELECT studentID, FullName, sat_score, rcd_updated FROM
student;
```

```
+-----------+------------------------+-----------+---------------------+
| studentID | FullName               | sat_score | rcd_updated         |
+-----------+------------------------+-----------+---------------------+
|         1 | Monique Davis          |       400 | 2017-08-16 15:34:50 |
|         2 | Teri Gutierrez         |       800 | 2017-08-16 15:34:50 |
|         3 | Spencer Pautier        |      1000 | 2017-08-16 15:34:50 |
|         4 | Louis Ramsey           |      1200 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene           |      1200 | 2017-08-16 15:34:50 |
|         6 | Sophie Freeman         |      1200 | 2017-08-16 15:34:50 |
|         7 | Edgar Frank "Ted" Codd |      2400 | 2017-08-16 15:35:33 |
|         8 | Donald D. Chamberlin   |      2400 | 2017-08-16 15:35:33 |
|         9 | Raymond F. Boyce       |      2400 | 2017-08-16 15:35:33 |
+-----------+------------------------+-----------+---------------------+
9 rows in set (0.00 sec)
```

Now, we'll repeat the SELECT query but we'll limit the rows returned using a WHERE statement.

```
STUDENT studentID, FullName, sat_score, recordUpdated
FROM student
WHERE (studentID BETWEEN 1 AND 5 OR studentID = 8)
        AND
        sat_score NOT IN (1000, 1400);
```

```
studentID | FullName             | sat_score | rcd_updated         |
+-----------+----------------------+-----------+---------------------+
|         1 | Monique Davis        |       400 | 2017-08-16 15:34:50 |
|         2 | Teri Gutierrez       |       800 | 2017-08-16 15:34:50 |
|         4 | Louis Ramsey         |      1200 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene         |      1200 | 2017-08-16 15:34:50 |
|         8 | Donald D. Chamberlin |      2400 | 2017-08-16 15:35:33 |
+-----------+----------------------+-----------+---------------------+
5 rows in set (0.00 sec)
```

UPDATE

To update a record in a table you use the UPDATE statement.
Use the WHERE condition to specify which records you want to update. It is possible to update one or more columns at a time. The syntax is:

```
UPDATE table_name
SET column1 = value1,
    column2 = value2, ...
WHERE condition;
```

Here is an example updating the Name of the record with Id 4:

```
UPDATE Person
SET Name = “Elton John”
WHERE Id = 4;
```

You can also update columns in a table by using values from other tables. Use the JOIN clause to get data from multiple tables. The syntax is:

```
UPDATE table_name1
SET table_name1.column1 = table_name2.columnA
    table_name1.column2 = table_name2.columnB
FROM table_name1
JOIN table_name2 ON table_name1.ForeignKey = table_name2.Key
```

GROUP BY

GROUP BY allows you to combine rows and aggregate data.
Here is the syntax of GROUP BY:

```
SELECT column_name, COUNT(*)
FROM table_name
GROUP BY column_name;
```

HAVING

HAVING allows you to filter the data aggregated by the GROUP BY clause so that the user gets a limited set of records to view.
Here is the syntax of HAVING:

```
SELECT column_name, COUNT(*)
FROM table_name
GROUP BY column_name
HAVING COUNT(*) > value;
```

AVG()
“Average” is used to calculate the average of a numeric column from the set of rows returned by a SQL statement.

Here is the syntax for using the function

```
SELECT groupingField, AVG(num_field)
FROM table1
GROUP BY groupingField
```

Here’s an example using the student table:

```
SELECT studentID, FullName, AVG(sat_score)
FROM student
GROUP BY studentID, FullName;
```

AS
AS allows us to rename a column or table using an alias.

```
SELECT user_only_num1 AS AgeOfServer, (user_only_num1 -
warranty_period) AS NonWarrantyPeriod
```

This results in output as below.

```
+-------------+-----------------------+
| AgeOfServer | NonWarrantyPeriod     |
+-------------+-----------------------+
|        36   |                    24 |
|        24  |                     12 |
|        61  |                     49 |
|        12  |                      0 |
|         6  |                     -6 |
|         0  |                    -12 |
|        36  |                     24 |
|        36  |                     24 |
|        24  |                     12 |
+------------+-----------------------+
```

we can also use AS to assign a name to a table to make it easier to reference in joins.

```
SELECT ord.product, ord.ord_number, ord.price,
cust.cust_name, cust.cust_number FROM customer_table AS cust

JOIN order_table AS ord ON cust.cust_number =
ord.cust_number
```

This results in output as below.

```
+------------+-----------+----------+----------------+-------------+
| product    | ord_number | price    | cust_name      | cust_number  |
+------------+-----------+----------+----------------+-------------+
|     RAM    |   12345    |       124 | John Smith     |  20          |
|     CPU    |   12346    |       212 | Mia X          |  22          |
|     USB    |   12347    |        49 | Elise Beth     |  21          |
|     Cable  |   12348    |         0 | Paul Fort      |  19          |
|     Mouse  |   12349    |        66 | Nats Back      |  15          |
|     Laptop |   12350    |       612 | Mel S          |  36          |
|     Keyboard|  12351    |        24 | George Z       |  95          |
|     Keyboard|  12352    |        24 | Ally B         |  55          |
|     Air    |   12353    |        12 | Maria Trust    |  11          |
+------------+-----------+----------+----------------+-------------+
```

ORDER BY

ORDER BY gives us a way to sort the result set by one or more of the items in the SELECT section. Here is an SQL sorting the students by FullName in descending order. The default sort order is ascending (ASC) but to sort in the opposite order (descending) you use DESC.

```
SELECT studentID, FullName, sat_score
FROM student
ORDER BY FullName DESC;
```

COUNT

COUNT will count the number of rows and return that count as a column in the result set.
Here are examples of what we would use COUNT for:

- Counting all rows in a table (no group by required)
- Counting the totals of subsets of data (requires a Group By section of the statement)

This SQL statement provides a count of all rows. we can give the resulting COUNT column a name using “AS”.

```
SELECT count(*) AS studentCount FROM student;
```

DELETE

DELETE is used to delete a record in a table.
we can delete all records of the table or just a few. Use
the WHERE condition to specify which records we want to delete. The syntax is:

```
DELETE FROM table_name
WHERE condition;
```

Here is an example deleting from the table Person the record with Id 3:

```
DELETE FROM Person
WHERE Id = 3;
```

INNER JOIN

JOIN, also called Inner Join, selects records that have matching values in two tables.

```
SELECT * FROM A x JOIN B y ON y.aId = x.Id
```

LEFT JOIN

A LEFT JOIN returns all rows from the left table, and the matched rows from the right table. Rows in the left table will be returned even if there was no match in the right table. The rows from the left table with no match in the right table will have null for right table values.

```
SELECT * FROM A x LEFT JOIN B y ON y.aId = x.Id
```

RIGHT JOIN

A RIGHT JOIN returns all rows from the right table, and the matched rows from the left table. Opposite of a left join, this will return all rows from the right table even where there is no match in the left table. Rows in the right table that have no match in the left table will have null values for left table columns.

```
SELECT * FROM A x RIGHT JOIN B y ON y.aId = x.Id
```

FULL OUTER JOIN

A FULL OUTER JOIN returns all rows for which there is a match in either of the tables. So if there are rows in the left table that do not have matches in the right table, those will be included. Also, if there are rows in the right table that do not have matches in the left table, those will be included.

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName
```

INSERT

INSERT is a way to insert data into a table.

```
INSERT INTO table_name (column_1, column_2, column_3)
VALUES (value_1, 'value_2', value_3);
```

LIKE

LIKE is used in a WHERE or HAVING (as part of the GROUP BY) to limit the selected rows to the items when a column has a certain pattern of characters contained in it.
This SQL will select students that have FullName starting with “Monique” or ending with “Greene”.

```
SELECT studentID, FullName, sat_score, rcd_updated
FROM student
WHERE
    FullName LIKE 'Monique%' OR
    FullName LIKE '%Greene';
```

```
SELECT studentID, FullName, sat_score, rcd_updated
FROM student
WHERE
```

```
    FullName LIKE 'Monique%' OR
    FullName LIKE '%Greene';
```

we can place NOT before LIKE to exclude the rows with the string pattern instead of selecting them. This SQL excludes records that contain “cer Pau” and “Ted” in the FullName column.

```
SELECT studentID, FullName, sat_score, rcd_updated
FROM student
WHERE FullName NOT LIKE '%cer Pau%' AND FullName NOT LIKE
'%"Ted"%';
```

```
+-----------+----------------------+-----------+---------------------+
| studentID | FullName             | sat_score | rcd_updated         |
+-----------+----------------------+-----------+---------------------+
|         1 | Monique Davis        |       400 | 2017-08-16 15:34:50 |
|         2 | Teri Gutierrez       |       800 | 2017-08-16 15:34:50 |
|         4 | Louis Ramsey         |      1200 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene         |      1200 | 2017-08-16 15:34:50 |
|         6 | Sophie Freeman       |      1200 | 2017-08-16 15:34:50 |
|         8 | Donald D. Chamberlin |      2400 | 2017-08-16 15:35:33 |
|         9 | Raymond F. Boyce     |      2400 | 2017-08-16 15:35:33 |
+-----------+----------------------+-----------+---------------------+
7 rows in set (0.00 sec)
```