The assessments include a number of written tests, including a SQL test.  The SQL test requires preparation, but is open book – you can bring and consult notes and text books if you wish.

**SQL Test interpret**

The SQL test is used by Linedata Services (BFT) to establish:

* An aptitude for relational databases and SQL
* Willingness and ability to learn something new in a short time

The performance of staff hired by Linedata Services (BFT) has been found to correlate well with performance of this test, regardless of whether SQL is regularly used in their work. The SQL test is often taken and passed well by economics or finance graduates with no prior computer training or experience, who started learning SQL just a few days before the test. Please note that the SQL test is "open book"; books and notes that you bring may be used during the test. However you are not allowed to use any laptops.

The SQL test is focused on (but not limited to) to writing various database queries and the **SELECT** statement, in particular:

1. **JOINS** (including outer-joins and self-joins)

2. **SUB-QUERIES** (covering both the "in" (sub-query) and "exists" (correlated sub-query) clauses)

3. **GROUP BY** (including the aggregate functions max (), min (), count(\*) and sum ()).

Set operators (union, minus and intersect) are not required.  A very basic knowledge of updates, inserts and deletes is also tested.  The test is primarily a test of the candidate's ability to apply these commands rather than recollection of syntax.  It tests standard SQL, not PL/SQL.

Candidates may use notes or a book that they supply; however no internet connection will be available. ***We strongly recommend that you prepare for this test, even if you are familiar with SQL.***

There are many training books on SQL. Good ones include "LAN Times Guide to SQL" by Groff and Weinberg, and "Practical SQL" by Judith Bowman. There are also many Internet sites with SQL tutorials on SQL, including:

<http://www.dcs.napier.ac.uk/~andrew/gisq/>

<http://www.sqlcourse.com/>

<http://www.sqlcourse2.com/>

<http://www.w3schools.com/sql/default.asp>

[http://www.sql.org](http://www.sql.org/)

Run the search engines to get the latest selection.

The other tests are not open book and require no preparation. The session will assess your problem solving, analytical and English language skills. Your scores in these – particularly the SQL test - will determine whether or not your application can be progressed further.

Table of Contents

Contents

[1) Select Statement 4](#_Toc503468841)

[2) Creating Tables 6](#_Toc503468842)

[**3)** **Inserting into a Table** 7](#_Toc503468843)

[4) Updating Records 8](#_Toc503468844)

[5) Deleting Records 9](#_Toc503468845)

[6) Drop a Table 9](#_Toc503468846)

[GROUP BY clause 10](#_Toc503468847)

[7) GROUP BY 10](#_Toc503468848)

[GROUP BY - Multiple Grouping Columns - What if? 10](#_Toc503468849)

[8) HAVING clause 11](#_Toc503468850)

[9) ORDER BY clause 12](#_Toc503468851)

[10) Combining Conditions & Boolean Operators 13](#_Toc503468852)

[11) IN & BETWEEN (NOT – can also use for LIKE) 13](#_Toc503468853)

[12) Mathematical Functions 15](#_Toc503468854)

[13 Table Joins, a must 17](#_Toc503468855)

[**Left outer join**[edit] 18](#_Toc503468856)

[**Right outer join**[edit] 19](#_Toc503468857)

[**Full outer join**[edit] 20](#_Toc503468858)

[Self Join 22](#_Toc503468859)

[14) Limit, Null 22](#_Toc503468860)

[15 Udacity Joints 22](#_Toc503468861)

[Joins: 23](#_Toc503468862)

[SUM,COUNT,MIN,MAX,AVG,MEDIAN(but this is not a SQL key word) 23](#_Toc503468863)

[16 Dates 24](#_Toc503468864)

[DATE\_TRUNC() to the rescue 24](#_Toc503468865)

[Date\_Part 25](#_Toc503468866)

[17) Case Statements 26](#_Toc503468867)

[18) Sub Query 27](#_Toc503468868)

[19) With Statement 28](#_Toc503468869)

[20) SQL TEXT OPERATIONS 29](#_Toc503468870)

[Here we looked at three new functions: LEFT, RIGHT, LENGTH 29](#_Toc503468871)

[SQL IN 30](#_Toc503468872)

[POSITION, STRPOS, LOWER, UPPER 30](#_Toc503468873)

[CONCAT and Piping || 31](#_Toc503468874)

[CAST 31](#_Toc503468875)

# Select Statement

The **select** statement is used to query the database and retrieve selected data that match the criteria that you specify. Here is the format of a simple select statement:

select "column1"

[,"column2",etc]

from "tablename"

[where "condition"];

[] = optional

The SELECT statement is used to query the database and retrieve selected data that match the criteria that you specify.

The SELECT statement has five main clauses to choose from, although, FROM is the only required clause. Each of the clauses have a vast selection of options, parameters, etc. The clauses will be listed below, but each of them will be covered in more detail later in the tutorial.

 Here is the format of the SELECT statement:

 SELECT [ALL | DISTINCT] column1[,column2] FROM table1[,table2] [WHERE "conditions"] [GROUP BY "column-list"] [HAVING "conditions] [ORDER BY "column-list" [ASC | DESC] ]

The column names that follow the select keyword determine which columns will be returned in the results. You can select as many column names that you'd like, or you can use a "\*" to select all columns.

The table name that follows the keyword **from** specifies the table that will be queried to retrieve the desired results.

The **where** clause (optional) specifies which data values or rows will be returned or displayed, based on the criteria described after the keyword **where**.

Conditional selections used in the **where** clause:

|  |  |
| --- | --- |
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| <> | Not equal to |
| LIKE | \*See note below |

The **LIKE** pattern matching operator can also be used in the conditional selection of the where clause. Like is a very powerful operator that allows you to select only rows that are "like" what you specify. The percent sign "%" can be used as a wild card to match any possible character that might appear before or after the characters specified. For example:

select first, last, city

from empinfo

where first LIKE 'Er%';

This SQL statement will match any first names that start with 'Er'. **Strings must be in single quotes.**

Or you can specify,

select first, last

from empinfo

where last LIKE '%s';

This statement will match any last names that end in a 's'.

select \* from empinfo

where first = 'Eric';

Enter the following sample select statements in the SQL Interpreter Form at the bottom of this page. Before you press "submit", write down your expected results. Press "submit", and compare the results.

select first, last, city from empinfo;

select last, city, age from empinfo

where age > 30;

select first, last, city, state from empinfo

where first LIKE 'J%';

select \* from empinfo;

select first, last, from empinfo

where last LIKE '%s';

select first, last, age from empinfo

where last LIKE '%illia%';

select \* from empinfo where first = 'Eric';

**ALL** and **DISTINCT** are keywords used to select either ALL (default) or the "distinct" or unique records in your query results. If you would like to retrieve just the unique records in specified columns, you can use the "DISTINCT" keyword. DISTINCT will discard the duplicate records for the columns you specified after the "SELECT" statement: For example:

SELECT DISTINCT age

FROM employee\_info;

This statement will return all of the unique ages in the employee\_info table.

ALL will display "all" of the specified columns including all of the duplicates. The ALL keyword is the default if nothing is specified.

# Creating Tables

The **create table** statement is used to create a new table. Here is the format of a simple **create table** statement:

create table "tablename"

("column1" "data type",

"column2" "data type",

"column3" "data type");

Format of create table if you were to use optional constraints:

create table "tablename"

("column1" "data type"

[constraint],

"column2" "data type"

[constraint],

"column3" "data type"

[constraint]);

[ ] = optional

**Note:** You may have as many columns as you'd like, and the constraints are optional.

**Example:**

create table employee

(first varchar(15),

last varchar(20),

age number(3),

address varchar(30),

city varchar(20),

state varchar(20));

To create a new table, enter the keywords **create table** followed by the table name, followed by an open parenthesis, followed by the first column name, followed by the data type for that column, followed by any optional constraints, and followed by a closing parenthesis. It is important to make sure you use an open parenthesis before the beginning table, and a closing parenthesis after the end of the last column definition. Make sure you seperate each column definition with a comma. All SQL statements should end with a ";".

The table and column names must start with a letter and can be followed by letters, numbers, or underscores - not to exceed a total of 30 characters in length. Do not use any SQL reserved keywords as names for tables or column names (such as "select", "create", "insert", etc).

Data types specify what the type of data can be for that particular column. If a column called "Last\_Name", is to be used to hold names, then that particular column should have a "varchar" (variable-length character) data type.

Here are the most common Data types:

|  |  |
| --- | --- |
| char(size) | Fixed-length character string. Size is specified in parenthesis. Max 255 bytes. |
| varchar(size) | Variable-length character string. Max size is specified in parenthesis. |
| number(size) | Number value with a max number of column digits specified in parenthesis. |
| date | Date value |
| number(size,d) | Number value with a maximum number of digits of "size" total, with a maximum number of "d" digits to the right of the decimal. |

What are constraints? When tables are created, it is common for one or more columns to have **constraints** associated with them. A constraint is basically a rule associated with a column that the data entered into that column must follow. For example, a "unique" constraint specifies that no two records can have the same value in a particular column. They must all be unique. The other two most popular constraints are "not null" which specifies that a column can't be left blank, and "primary key". A "primary key" constraint defines a unique identification of each record (or row) in a table. All of these and more will be covered in the future Advanced release of this Tutorial. Constraints can be entered in this SQL interpreter, however, they are not supported in this Intro to SQL tutorial & interpreter. They will be covered and supported in the future release of the Advanced SQL tutorial - that is, if "response" is good.

1. **Inserting into a Table**

The **insert** statement is used to insert or add a row of data into the table.

To insert records into a table, enter the key words **insert into** followed by the table name, followed by an open parenthesis, followed by a list of column names separated by commas, followed by a closing parenthesis, followed by the keyword **values**, followed by the list of values enclosed in parenthesis. The values that you enter will be held in the rows and they will match up with the column names that you specify. Strings should be enclosed in single quotes, and numbers should not.

insert into "tablename"  
 (first\_column,...last\_column)  
 values (first\_value,...last\_value);

In the example below, the column name first will match up with the value 'Luke', and the column name state will match up with the value 'Georgia'.

**Example:**

insert into employee  
 (first, last, age, address, city, state)  
 values ('Luke', 'Duke', 45, '2130 Boars Nest',   
 'Hazard Co', 'Georgia');

**Note:** All strings should be enclosed between **single** quotes: 'string'

# Updating Records

The **update** statement is used to update or change records that match a specified criteria. This is accomplished by carefully constructing a where clause.

update "tablename"  
set "columnname" =   
 "newvalue"  
 [,"nextcolumn" =   
 "newvalue2"...]  
where "columnname"   
 OPERATOR "value"   
 [and|or "column"   
 OPERATOR "value"];  
  
 [] = optional

*[The above example was line wrapped for better viewing on this Web page.]*

Examples:

update phone\_book  
 set area\_code = 623  
 where prefix = 979;  
  
update phone\_book  
 set last\_name = 'Smith', prefix=555, suffix=9292  
 where last\_name = 'Jones';  
  
update employee  
 set age = age+1  
 where first\_name='Mary' and last\_name='Williams';

# Deleting Records

The delete statement is used to delete records or rows from the table.

delete from "tablename"

where "columnname"

OPERATOR "value"

[and|or "column"

OPERATOR "value"];

[ ] = optional

[The above example was line wrapped for better viewing on this Web page.]

Examples:

delete from employee;

Note: if you leave off the where clause, all records will be deleted!

delete from employee

where lastname = 'May';

delete from employee

where firstname = 'Mike' or firstname = 'Eric';

To delete an entire record/row from a table, enter "delete from" followed by the table name, followed by the where clause which contains the conditions to delete. If you leave off the where clause, all records will be deleted.

# Drop a Table

The **drop table** command is used to delete a table and all rows in the table.

To delete an entire table including all of its rows, issue the **drop table** command followed by the tablename. **drop table** is different from deleting all of the records in the table. Deleting all of the records in the table leaves the table including column and constraint information. Dropping the table removes the table definition as well as all of its rows.

drop table "tablename"

**Example:**

drop table myemployees\_ts0211;

# GROUP BY clause

The GROUP BY clause will gather all of the rows together that contain data in the specified column(s) and will allow aggregate functions to be performed on the one or more columns. This can best be explained by an example:

# **GROUP BY**

clause syntax: sum, min,max,count,average

SELECT column1,   
SUM(column2)  
   
FROM "list-of-tables"  
   
GROUP BY "column-list";

Let's say you would like to retrieve a list of the highest paid salaries in each dept:

SELECT max(salary), dept  
   
FROM employee   
   
GROUP BY dept;

This statement will select the maximum salary for the people in each unique department. Basically, the salary for the person who makes the most in each department will be displayed. Their, salary and their department will be returned.

# GROUP BY - Multiple Grouping Columns - What if?

What if you ALSO want to display their lastname for the query below:

SELECT max(salary), dept  
FROM employee  
GROUP BY dept;

What you'll need to do is:

SELECT lastname, max(salary), dept  
FROM employee  
GROUP BY dept, lastname;

This is a called "multiple grouping columns".

e.g.

SELECT customerid, count(customerid), sum(price\*quantity)

FROM items\_ordered

GROUP BY customerid;

# HAVING clause

The HAVING clause allows you to specify conditions on the rows for each group - in other words, which rows should be selected will be based on the conditions you specify. The HAVING clause should follow the GROUP BY clause if you are going to use it.

**HAVING** clause syntax:

SELECT column1,   
SUM(column2)  
  
FROM "list-of-tables"  
  
GROUP BY "column-list"  
  
HAVING "condition";

HAVING can best be described by example. Let's say you have an employee table containing the employee's name, department, salary, and age. If you would like to select the average salary for each employee in each department, you could enter:

SELECT dept, avg(salary)  
  
  
FROM employee  
  
GROUP BY dept;

But, let's say that you want to ONLY calculate & display the average if their salary is over 20000:

SELECT dept, avg(salary)  
  
FROM employee  
  
GROUP BY dept  
  
HAVING avg(salary) > 20000;

# ORDER BY clause

ORDER BY is an optional clause which will allow you to display the results of your query in a sorted order (either ascending order or descending order) based on the columns that you specify to order by.

**ORDER BY** clause syntax:

SELECT column1, SUM(column2) FROM "list-of-tables" ORDER BY "column-list" [ASC | DESC];

[ ] = optional

This statement will select the employee\_id, dept, name, age, and salary from the employee\_info table where the dept equals 'Sales' and will list the results in Ascending (default) order based on their Salary.

ASC = Ascending Order - default  
  
DESC = Descending Order

For example:

SELECT employee\_id, dept, name, age, salary FROM employee\_info WHERE dept = 'Sales' ORDER BY salary;

If you would like to order based on multiple columns, you must seperate the columns with commas. For example:

SELECT employee\_id, dept, name, age, salary  
  
  
FROM employee\_info  
  
WHERE dept = 'Sales'  
  
ORDER BY salary, age DESC;

# Combining Conditions & Boolean Operators

The AND operator can be used to join two or more conditions in the WHERE clause. Both sides of the AND condition must be true in order for the condition to be met and for those rows to be displayed.

SELECT column1,   
SUM(column2)  
  
FROM "list-of-tables"  
  
  
WHERE "condition1" AND   
"condition2";

The OR operator can be used to join two or more conditions in the WHERE clause also. However, **either** side of the OR operator can be true and the condition will be met - hence, the rows will be displayed. With the OR operator, either side can be true or both sides can be true.

For example:

SELECT employeeid, firstname, lastname, title, salary  
  
  
FROM employee\_info  
  
WHERE salary >= 45000.00 AND title = 'Programmer';

# IN & BETWEEN (NOT – can also use for LIKE)

SELECT col1, SUM(col2)  
  
FROM "list-of-tables"  
  
WHERE col3 IN   
 (list-of-values);  
  
SELECT col1, SUM(col2)  
  
  
FROM "list-of-tables"  
  
WHERE col3 BETWEEN value1   
AND value2;

The IN conditional operator is really a set membership test operator. That is, it is used to test whether or not a value (stated before the keyword IN) is "in" the list of values provided after the keyword **IN**.

For example:

SELECT employeeid, lastname, salary  
  
FROM employee\_info  
  
  
WHERE lastname IN ('Hernandez', 'Jones', 'Roberts', 'Ruiz');

 SELECT name, primary\_poc, sales\_rep\_id

FROM accounts

where name NOT IN ('Walmart', 'Target', 'Nordstrom')

LIMIT 10;

SELECT \*

FROM accounts

WHERE name NOT LIKE 'C%'

LIMIT 100;

This statement will select the employeeid, lastname, salary from the employee\_info table where the lastname is equal to either: Hernandez, Jones, Roberts, or Ruiz. It will return the rows if it is ANY of these values.

You can also use **NOT IN** to exclude the rows in your list.

The BETWEEN conditional operator is used to test to see whether or not a value (stated before the keyword BETWEEN) is "between" the two values stated after the keyword BETWEEN.

For example:

SELECT employeeid, age, lastname, salary  
  
FROM employee\_info  
   
WHERE age BETWEEN 30 AND 40;

This statement will select the employeeid, age, lastname, and salary from the employee\_info table where the age is between 30 and 40 (including 30 and 40).

This statement can also be rewritten without the BETWEEN operator:

SELECT employeeid, age, lastname, salary  
  
FROM employee\_info  
  
WHERE age >= 30 AND age <= 40;

You can also use **NOT BETWEEN** to exclude the values between your range.

# Mathematical Functions

Standard ANSI SQL-92 supports the following first four basic arithmetic operators:

|  |  |
| --- | --- |
| + | addition |
| - | subtraction |
| \* | multiplication |
| / | division |
| % | modulo |

The modulo operator determines the integer remainder of the division. This operator is not ANSI SQL supported, however, most databases support it. The following are some more useful mathematical functions to be aware of since you might need them. These functions are not standard in the ANSI SQL-92 specs, therefore they may or may not be available on the specific RDBMS that you are using. However, they were available on several major database systems that I tested. They WILL work on this tutorial.

|  |  |
| --- | --- |
| ABS(x) | returns the absolute value of x |
| SIGN(x) | returns the sign of input x as -1, 0, or 1 (negative, zero, or positive respectively) |
| MOD(x,y) | modulo - returns the integer remainder of x divided by y (same as x%y) |
| FLOOR(x) | returns the largest integer value that is less than or equal to x |
| CEILING(x) or CEIL(x) | returns the smallest integer value that is greater than or equal to x |
| POWER(x,y) | returns the value of x raised to the power of y |
| ROUND(x) | returns the value of x rounded to the nearest whole integer |
| ROUND(x,d) | returns the value of x rounded to the number of decimal places specified by the value d |
| SQRT(x) | returns the square-root value of x |

For example:

SELECT round(salary), firstname  
  
FROM employee\_info

SELECT quantity, price, price\*quantity (this will be a derived field)

FROM orders;

SELECT id, account\_id, poster\_amt\_usd/(standard\_amt\_usd+gloss\_amt\_usd+ +poster\_amt\_usd+1) AS post\_per

FROM orders

LIMIT 10;

This statement will select the salary rounded to the nearest whole value and the firstname from the employee\_info table.

# 13 Table Joins, a must

All of the queries up until this point have been useful with the exception of one major limitation - that is, you've been selecting from only one table at a time with your SELECT statement. It is time to introduce you to one of the most beneficial features of SQL & relational database systems - the "**Join**". To put it simply, the "Join" makes relational database systems "relational".

Joins allow you to link data from two or more tables together into a single query result--from one single SELECT statement.

A "Join" can be recognized in a SQL SELECT statement if it has more than one table after the FROM keyword.

For example:

SELECT "list-of-columns"  
  
FROM *table1*,*table2*  
  
WHERE "search-condition(s)"

Notice how each of the tables have a common "cusomer\_number" column. This column, which contains the unique customer number will be used to **JOIN** the two tables. Using the two new tables, let's say you would like to select the customer's name, and items they've purchased. Here is an example of a join statement to accomplish this:

SELECT customer\_info.firstname, customer\_info.lastname, purchases.item  
  
FROM customer\_info, purchases  
  
WHERE customer\_info.customer\_number = purchases.customer\_number;

This particular "Join" is known as an "Inner Join" or "Equijoin". This is the most common type of "Join" that you will see or use.

Notice that each of the colums are always preceeded with the table name and a period. This isn't always required, however, it IS good practice so that you wont confuse which colums go with what tables. It is required if the name column names are the same between the two tables. I recommend preceeding all of your columns with the table names when using joins.

**Note: The syntax described above will work with most Database Systems -including the one with this tutorial. However, in the event that this doesn't work with yours, please check your specific database documentation.**

Although the above will probably work, here is the ANSI SQL-92 syntax specification for an Inner Join using the preceding statement above that you might want to try:

SELECT customer\_info.firstname, customer\_info.lastname, purchases.item  
  
FROM customer\_info INNER JOIN purchases  
  
ON customer\_info.customer\_number = purchases.customer\_number;

Another example:

SELECT employee\_info.employeeid, employee\_info.lastname, employee\_sales.comission  
  
FROM employee\_info, employee\_sales  
  
WHERE employee\_info.employeeid = employee\_sales.employeeid;

This statement will select the employeeid, lastname (from the employee\_info table), and the comission value (from the employee\_sales table) for all of the rows where the employeeid in the employee\_info table matches the employeeid in the employee\_sales table.

**Left outer join**[[edit](https://en.wikipedia.org/w/index.php?title=Join_(SQL)&action=edit&section=7)]

The result of a *left outer join* (or simply **left join**) for tables A and B always contains all rows of the "left" table (A), even if the join-condition does not find any matching row in the "right" table (B). This means that if the ON clause matches 0 (zero) rows in B (for a given row in A), the join will still return a row in the result (for that row)—but with NULL in each column from B. A **left outer join** returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link column.

For example, this allows us to find an employee's department, but still shows employees that have not been assigned to a department (contrary to the inner-join example above, where unassigned employees were excluded from the result).

Example of a left outer join (the **OUTER** keyword is optional), with the additional result row (compared with the inner join) italicized:

**SELECT** \*

**FROM** employee

**LEFT** **OUTER** **JOIN** department **ON** employee.DepartmentID = department.DepartmentID;

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Jones | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| Robinson | 34 | Clerical | 34 |
| Smith | 34 | Clerical | 34 |
| *Williams* | NULL | NULL | NULL |
| Heisenberg | 33 | Engineering | 33 |

**Right outer join**[[edit](https://en.wikipedia.org/w/index.php?title=Join_(SQL)&action=edit&section=9)]

A **right outer join** (or **right join**) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those rows that have no match in B.

A right outer join returns all the values from the right table and matched values from the left table (NULL in the case of no matching join predicate). For example, this allows us to find each employee and his or her department, but still show departments that have no employees.

Below is an example of a right outer join (the **OUTER** keyword is optional), with the additional result row italicized:

**SELECT** \*

**FROM** employee **RIGHT** **OUTER** **JOIN** department

**ON** employee.DepartmentID = department.DepartmentID;

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Smith | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Robinson | 34 | Clerical | 34 |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| NULL | NULL | *Marketing* | *35* |

Right and left outer joins are functionally equivalent. Neither provides any functionality that the other does not, so right and left outer joins may replace each other as long as the table order is switched.

**Full outer join**[[edit](https://en.wikipedia.org/w/index.php?title=Join_(SQL)&action=edit&section=10)]

Conceptually, a **full outer join** combines the effect of applying both left and right outer joins. Where rows in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those rows that do match, a single row will be produced in the result set (containing columns populated from both tables).

For example, this allows us to see each employee who is in a department and each department that has an employee, but also see each employee who is not part of a department and each department which doesn't have an employee.

Example of a full outer join (the **OUTER** keyword is optional):

**SELECT** \*

**FROM** employee **FULL** **OUTER** **JOIN** department

**ON** employee.DepartmentID = department.DepartmentID;

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Smith | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Robinson | 34 | Clerical | 34 |
| *Williams* | NULL | NULL | NULL |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| NULL | NULL | *Marketing* | *35* |

## Self Join

|  |  |  |
| --- | --- | --- |
| Well, one classic example is where you wanted to get a list of employees and their immediate managers:  select e.employee as employee, b.employee as boss  from emptable e, emptable b  where e.manager\_id = b.empolyee\_id  order by 1  It's basically used where there is any relationship between rows stored in the same table.   * employees. * multi-level marketing. * machine parts.   And so on...   |  |  | | --- | --- | | [share](https://stackoverflow.com/a/3362068/8839930)[edit](https://stackoverflow.com/posts/3362068/edit) | answered Jul 29 '10 at 11:39  [[https://i.stack.imgur.com/vXG1F.png?s=32&g=1](https://stackoverflow.com/users/14860/paxdiablo)](https://stackoverflow.com/users/14860/paxdiablo)  [paxdiablo](https://stackoverflow.com/users/14860/paxdiablo)  **557k**14311061545 | |
|  | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | | 6 |  | | Boo for non-ANSI syntax and using ordinal in the ORDER BY clause rather than a column name. – [RedFilter](https://stackoverflow.com/users/39430/redfilter)[Jul 29 '10 at 12:06](https://stackoverflow.com/questions/3362038/what-is-self-join-and-when-would-you-use-it#comment3491887_3362068) | | |  |  | | --- | --- | | 13 |  | | Boo right back at you for missing the point of the answer :-) I tend to use the simplest syntax that works since the good DBMS' won't let that affect the performance. – [paxdiablo](https://stackoverflow.com/users/14860/paxdiablo) [Jul 29 '10 at 12:34](https://stackoverflow.com/questions/3362038/what-is-self-join-and-when-would-you-use-it#comment3492155_3362068) | | |  |  | | --- | --- | |  |  | | best answer so far... :) – [Sritam Jagadev](https://stackoverflow.com/users/4141569/sritam-jagadev) [May 29 at 9:45](https://stackoverflow.com/questions/3362038/what-is-self-join-and-when-would-you-use-it#comment75487998_3362068) | |

# 14) Limit, Null

e.g. SELECT \*

FROM accounts

LIMIT 10

The query will show only the first 10 rows

WHERE column IS NULL

WHERE column IS NOT NULL

# 15 Udacity Joins

## Joins:

Inner join:

SELECT orders.\*

FROM orders

JOIN accounts

ON orders.account\_id = accounts.id;

Multiple table join example:

**SELECT** \*

**FROM** web\_events

**JOIN** accounts

**ON** web\_events.account\_id = accounts.**id**

**JOIN** orders

**ON** accounts.**id** = orders.account\_id

Aliases – ‘as’ is optional:

SELECT a.name,a.primary\_poc,w.channel,w.occurred\_at

FROM accounts as a

JOIN web\_events as w

ON w.account\_id = a.id

where a.name = 'Walmart';

Alisases without as and with column names, also you can add logic to the ‘ON’ using an ‘AND’:

SELECT r.name region, s.name rep, a.name account

FROM sales\_reps s

JOIN region r

ON s.region\_id = r.id

JOIN accounts a

ON a.sales\_rep\_id = s.id

AND r.name = 'Midwest'

ORDER BY a.name;

## SUM,COUNT,MIN,MAX,AVG,MEDIAN(but this is not a SQL key word)

To get the rows of a table:

SELECT COUNT(\*)

FROM table

To get the rows of a table – you can use count to get also column name counts – nulls do not get counted :

SELECT COUNT(column name)

FROM table

You can use sum to get also column name aggregations – nulls get counted as zero:

SELECT SUM(column name)

FROM table

You can use MIN,MAX,AVG to get also column name aggregations, for AVG null do not get counted when calculating the average:

SELECT MIN,MAX,AVG(column name)

FROM table

Median: Since there are 6912 orders - we want the average of the 3457 and 3456 order amounts when ordered. This is the average of 2483.16 and 2482.55. This gives the median of **2482.855**. This obviously isn't an ideal way to compute. If we obtain new orders, we would have to change the limit. SQL didn't even calculate the median for us. The above used a SUBQUERY, but you could use any method to find the two necessary values, and then you just need the average of them.

1. **SELECT** \*
2. **FROM** (**SELECT** total\_amt\_usd
3. **FROM** orders
4. **ORDER** **BY** total\_amt\_usd
5. **LIMIT** 3457) **AS** Table1
6. **ORDER** **BY** total\_amt\_usd **DESC**
7. **LIMIT** 2;

## JOINS with additional operators

You can add extra conditions to the ON statement, note that strings columns cannot be matached with ‘=’

SELECT a.name account\_name, a.primary\_poc primary\_poc, s.name sales\_rep

FROM accounts a

JOIN sales\_reps s

ON s.id = a.sales\_rep\_id AND (a.primary\_poc < s.name OR a.primary\_poc > s.name);

## SELF JOINS:

Example:

SELECT we1.id AS we\_id,

we1.account\_id AS we1\_account\_id,

we1.occurred\_at AS we1\_occurred\_at,

we1.channel AS we1\_channel,

we2.id AS we2\_id,

we2.account\_id AS we2\_account\_id,

we2.occurred\_at AS we2\_occurred\_at,

we2.channel AS we2\_channel

FROM web\_events we1

LEFT JOIN web\_events we2

ON we1.account\_id = we2.account\_id

AND we1.occurred\_at > we2.occurred\_at

AND we1.occurred\_at <= we2.occurred\_at + INTERVAL '28 days'

ORDER BY we1.account\_id, we2.occurred\_at

## UNIONS

UNION only appends distinct values. More specifically, when you use UNION, the dataset is appended, and any rows in the appended table that are exactly identical to rows in the first table are dropped. If you’d like to append all the values from the second table, use UNION ALL. You’ll likely use UNION ALL far more often than UNION.

WITH account\_union AS

(SELECT \*

FROM accounts a1

UNION ALL

SELECT \*

FROM Accounts a2)

SELECT count(\*)

FROM account\_union;

# 16 Dates

## DATE\_TRUNC() to the rescue

To remove the unwanted detail of a timestamp, feed it into the DATE\_TRUNC(‘[interval]’, time\_column) function. time\_column is the database column that contains the timestamp you’d like to round, and ‘[interval]’ dictates your desired precision level. You can round off a timestamp to one of these units of time:

* microsecond
* millisecond
* second
* minute
* hour
* day
* week
* month
* quarter
* year
* decade
* century
* millenium

If the above timestamp were rounded down to 'day', the result is:

2015-10-06T00:00:00.000Z

If it were rounded down to 'minute', it would look like this:

2015-10-06T11:54:00.000Z

Likewise, ‘second’ rounds down to the nearest second, ‘hour’ down to the nearest hour, and so on. ‘week’ rounds down to that Monday’s date.

Example:

SELECT DATE\_TRUNC('month',o.occurred\_at) AS Year, SUM(o.total\_amt\_usd) AS Sales

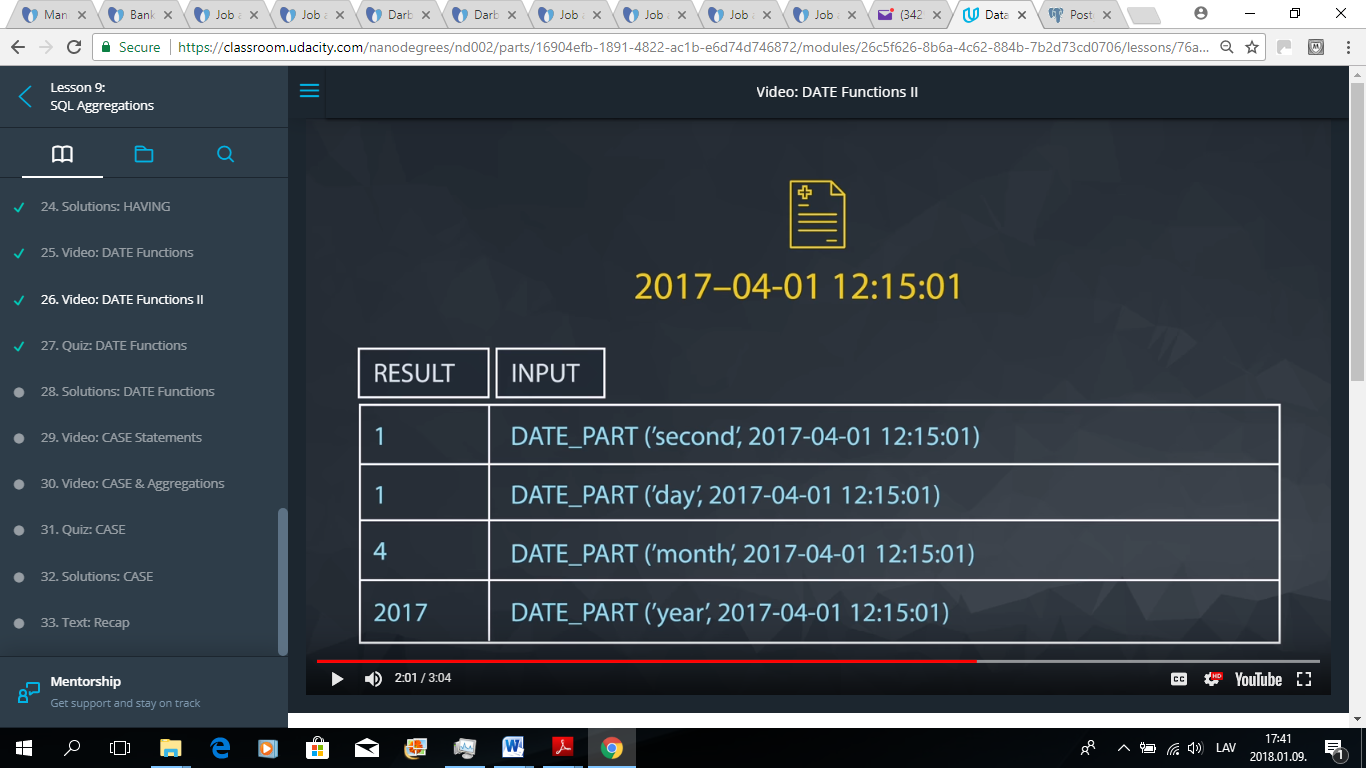
FROM orders o

GROUP BY 1

ORDER BY 2 DESC;

## Date\_Part

More documentation here: <https://www.postgresql.org/docs/9.1/static/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT>



Example – used column numbers for GROUP BY and ORDER BY:

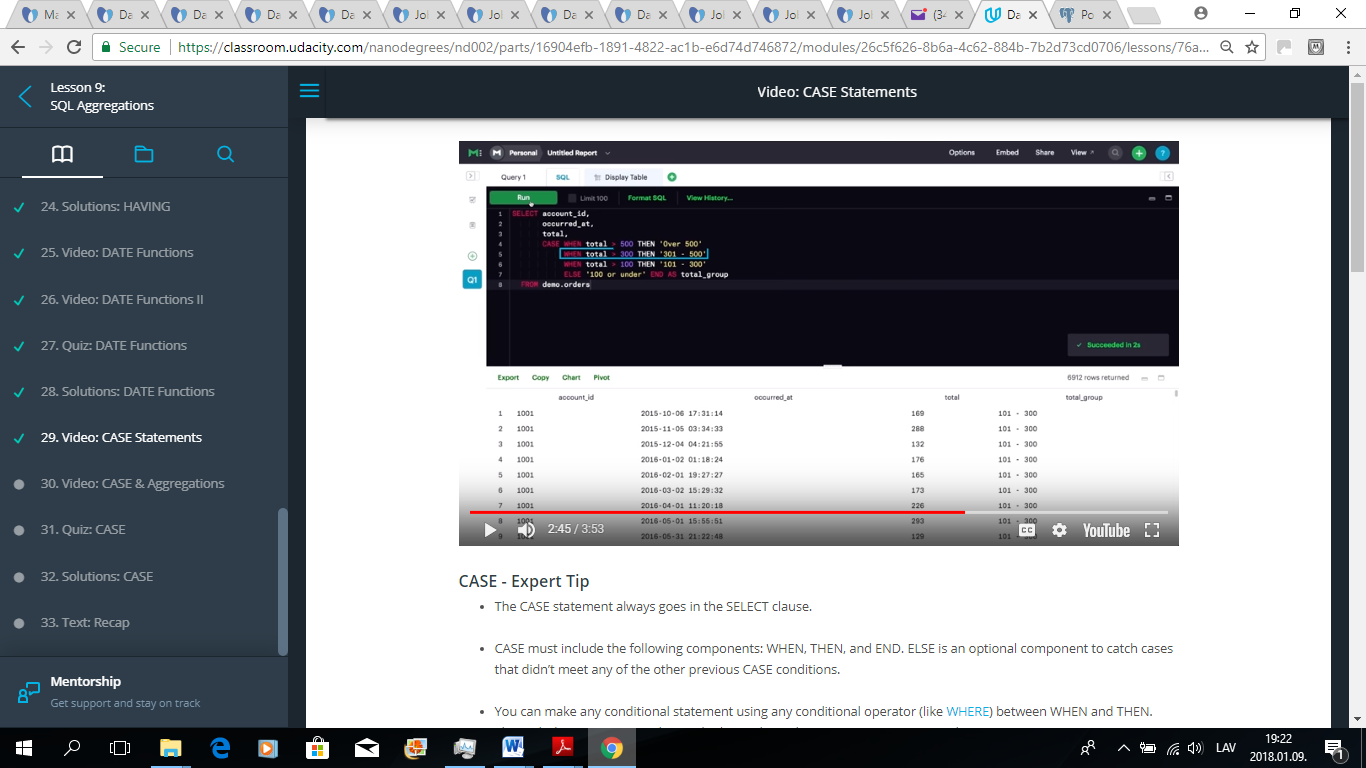
SELECT DATE\_PART('year',o.occurred\_at) AS Year, SUM(o.total\_amt\_usd) AS Sales

FROM orders o

GROUP BY 1

ORDER BY 2 DESC;

# 17) Case Statements



Example:

SELECT a.name AS Account , SUM(o.total\_amt\_usd) AS Spent,

CASE WHEN SUM(o.total\_amt\_usd) > 200000 THEN '> 200 000'

WHEN SUM(o.total\_amt\_usd) < 200000 AND SUM(o.total\_amt\_usd) > 100000 THEN '> 100 000 and < 200 000'

WHEN SUM(o.total\_amt\_usd) < 100000 THEN '< 100 000'

END AS Spend\_Group

FROM accounts a

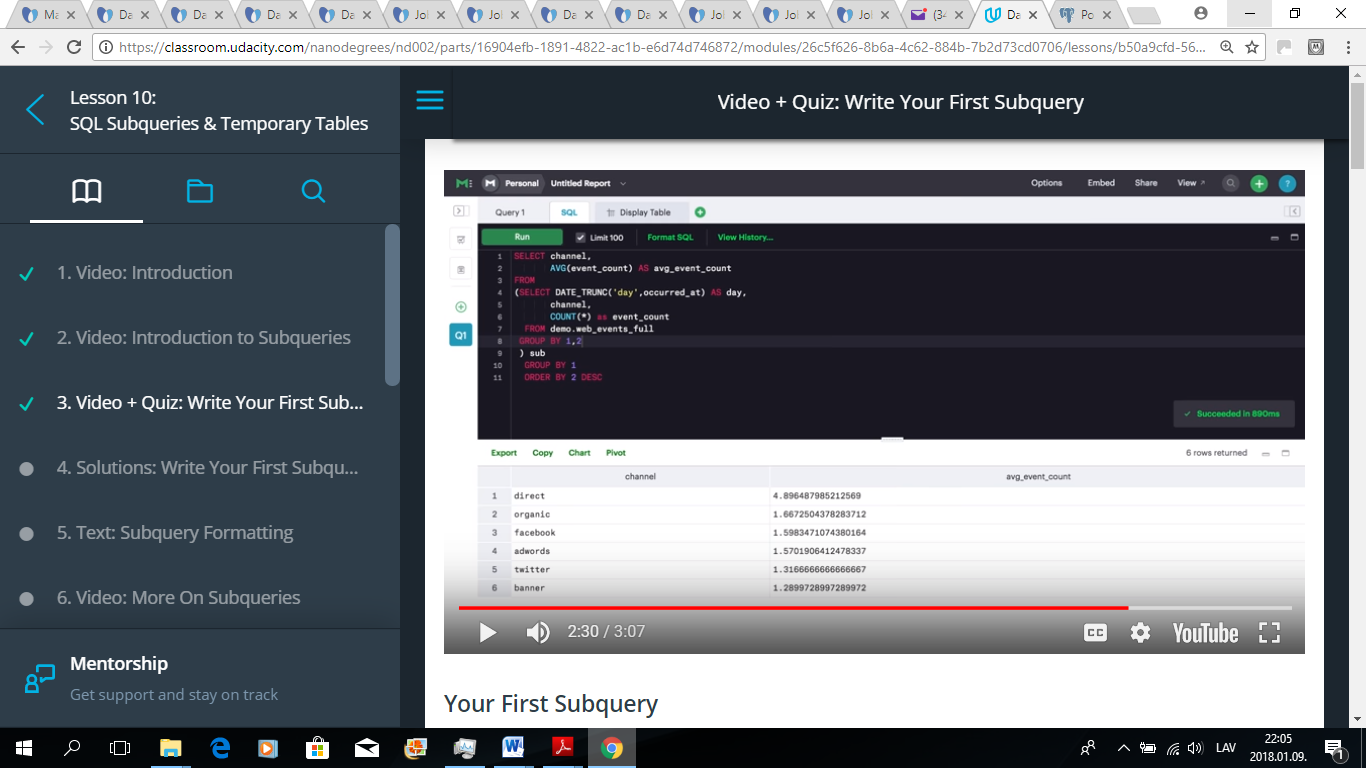
JOIN orders o

ON a.id = o.account\_id

GROUP BY Account

ORDER BY Spent DESC;

# 18) Sub Query



Example #1:

SELECT channel,day,AVG(events\_per\_day) AS averages\_events\_per\_day

FROM

(SELECT w.channel AS channel, DATE\_TRUNC('day',w.occurred\_at) AS day, COUNT(w.\*) AS events\_per\_day

FROM web\_events w

GROUP BY 1,2) AS sub

GROUP BY 1,2

ORDER BY 2,1

Example #2, if you use the subquerty to get a where value, do not use an alias for the subquery :

SELECT AVG(o.standard\_qty) AS avg\_standard\_qty, AVG(o.gloss\_qty) AS avg\_gloss\_qty, AVG(o.poster\_qty) as avg\_poster\_qty

FROM ORDERS O

WHERE DATE\_TRUNC('month',o.occurred\_at) =

(SELECT MIN(DATE\_TRUNC('month',o.occurred\_at))

FROM orders o);

# 19) With Statement

A with statement allow you to make temporary tables. This can make sub queries easier to follow – less nesting. Note that for multiple with statement then need to be separated by ‘,’ except for the last one.

Example:

WITH events AS (

**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,

channel, **COUNT**(\*) **as** **events**

**FROM** web\_events

**GROUP** **BY** 1,2)

**SELECT** channel, **AVG**(**events**) **AS** average\_events

**FROM** **events**

**GROUP** **BY** channel

**ORDER** **BY** 2 **DESC**;

WITH table1 AS (

**SELECT** \*

**FROM** web\_events),

table2 **AS** (

**SELECT** \*

**FROM** accounts)

**SELECT** \*

**FROM** table1

**JOIN** table2

**ON** table1.account\_id = table2.**id**;

a

# 20) SQL TEXT OPERATIONS

<https://www.postgresql.org/docs/8.1/static/functions-string.html>

## Here we looked at three new functions: **LEFT,** **RIGHT,** **LENGTH**

You can also sort and perform test operations on alfpha numerics. I would not trust mathematical operations.

**LEFT** pulls a specified number of characters for each row in a specified column starting at the beginning (or from the left). As you saw here, you can pull the first three digits of a phone number using **LEFT(phone\_number, 3)**.

**RIGHT** pulls a specified number of characters for each row in a specified column starting at the end (or from the right). As you saw here, you can pull the last eight digits of a phone number using **RIGHT(phone\_number, 8)**.

**LENGTH** provides the number of characters for each row of a specified column. Here, you saw that we could use this to get the length of each phone number as **LENGTH(phone\_number)**.

Examples, second one is to get exaction positions:

SELECT a.website, right(a.website,3) ext

FROM accounts a;

SELECT a.website, right(left(a.website,5),1) AS first

FROM accounts a;

## SQL IN

SQL also has an IN statement. Here is an example:

SELECT first,

CASE WHEN first IN ('0','1','2','3','4','5','6','7', '8','9') THEN 'Number'

ELSE 'Letter'

END AS first\_type

FROM first\_letter

ORDER BY 2 DESC;

## **POSITION**, **STRPOS**, **LOWER**, **UPPER**

**POSITION** takes a character and a column, and provides the index where that character is for each row. The index of the first position is 1 in SQL. If you come from another programming language, many begin indexing at 0. Here, you saw that you can pull the index of a comma as **POSITION(',' IN city\_state)**.

**STRPOS** provides the same result as **POSITION**, but the syntax for achieving those results is a bit different as shown here: **STRPOS(city\_state, ',')**.

Note, both **POSITION** and **STRPOS** are case sensitive, so looking for **A** is different than looking for **a**.

Therefore, if you want to pull an index regardless of the case of a letter, you might want to use **LOWER** or **UPPER** to make all of the characters lower or uppercase.

In the examples, there is also the **SUBSTR** example.

Have questions? Head to the [**forums**](https://forums.udacity.com/?forum_path=c/nd002-data-analyst-broadcast)

Examples:

SELECT a.primary\_poc, LEFT(a.primary\_poc,POSITION(' ' IN a.primary\_poc)) as first\_name, RIGHT(a.primary\_poc,LENGTH(a.primary\_poc)-POSITION(' ' IN a.primary\_poc)) as last\_name

FROM accounts a;

SELECT s.name, LEFT(s.name,STRPOS(s.name,' ')) AS first\_name, RIGHT(s.name,LENGTH(s.name) - STRPOS(s.name,' ')) AS last\_name

FROM sales\_reps s;

SELECT date orig\_date, (SUBSTR(date, 7, 4) || '-' || LEFT(date, 2) || '-' || SUBSTR(date, 4, 2))::DATE new\_date

FROM sf\_crime\_data

LIMIT 10;

## **CONCAT** and Piping ||

Each of these will allow you to combine columns together across rows. In this video, you saw how first and last names stored in separate columns could be combined together to create a full name: **CONCAT(first\_name, ' ', last\_name)** or with piping as **first\_name || ' ' || last\_name**. Note that the example has the ‘REPLACE’ command to remove spaces.

Example:

SELECT a.primary\_poc, LEFT(a.primary\_poc,POSITION(' ' IN a.primary\_poc)) as first\_name, RIGHT(a.primary\_poc,LENGTH(a.primary\_poc)-POSITION(' ' IN a.primary\_poc)) as last\_name, a.name AS company,

CONCAT(LEFT(a.primary\_poc,POSITION(' ' IN a.primary\_poc)-1),'.',RIGHT(a.primary\_poc,LENGTH(a.primary\_poc)-POSITION(' ' IN a.primary\_poc)),'@',REPLACE(a.name,' ',''),'.com') AS e\_mail

FROM accounts a;

## ****CAST****

**TO\_DATE**

**CAST**

Casting with ::

**DATE\_PART('month', TO\_DATE(month, 'month'))** here changed a month name into the number associated with that particular month.

Then you can change a string to a date using **CAST**. **CAST** is actually useful to change lots of column types. Commonly you might be doing as you saw here, where you change a string to a date using **CAST(date\_column AS DATE)**. However, you might want to make other changes to your columns in terms of their data types. You can see other examples [**here**](http://www.postgresqltutorial.com/postgresql-cast/).

In this example, you also saw that instead of **CAST(date\_column AS DATE)**, you can use **date\_column::DATE**.

Expert Tip

Most of the functions presented in this lesson are specific to strings. They won’t work with dates, integers or floating-point numbers. However, using any of these functions will automatically change the data to the appropriate type.

**LEFT**, **RIGHT**, and **TRIM** are all used to select only certain elements of strings, but using them to select elements of a number or date will treat them as strings for the purpose of the function. Though we didn't cover **TRIM** in this lesson explicitly, it can be used to remove characters from the beginning and end of a string. This can remove unwanted spaces at the beginning or end of a row that often happen with data being moved from Excel or other storage systems.

There are a number of variations of these functions, as well as several other string functions not covered here. Different databases use subtle variations on these functions, so be sure to look up the appropriate database’s syntax if you’re connected to a private database.The [**Postgres literature**](http://www.postgresql.org/docs/9.1/static/functions-string.html) contains a lot of the related functions.

Example:

WITH new\_date AS

(SELECT s.date,

RIGHT(LEFT(s.date,10),4) AS yyyy,

RIGHT(LEFT(s.date,5),2) AS dd,

LEFT(s.date,2) as mm

FROM sf\_crime\_data s

LIMIT 10)

SELECT n.date, CAST(CONCAT(n.yyyy,'-',n.mm,'-',n.dd) AS DATE) AS new\_date

FROM new\_date n;

## COALESCE

In this video, you learned about how to use **COALESCE** to work with NULL values. Unfortunately, our dataset does not have the **NULL** values that were fabricated in this dataset, so you will work through a different example in the next concept to get used to the **COALESCE** function.

In general, **COALESCE** returns the first non-NULL value passed for each row. Hence why the video used **no\_poc** if the value in the row was NULL.

Example:

SELECT COALESCE(a.id,a.id) id,a.name,a.website,a.lat,a.long,a.primary\_poc,a.sales\_rep\_id,COALESCE(o.standard\_qty,0) standard\_qty,COALESCE(o.gloss\_qty,0) gloss\_qty,COALESCE(o.poster\_qty,0) poster\_qty,COALESCE(o.total,0) total,COALESCE(o.standard\_amt\_usd,0) standard\_amt\_usd,COALESCE(o.gloss\_amt\_usd,0) gloss\_amt\_usd,COALESCE(o.poster\_amt\_usd,0) poster\_amt\_usd,COALESCE(o.total\_amt\_usd,0) total\_amt\_usd

FROM accounts a

LEFT JOIN orders o

ON a.id = o.account\_id

WHERE o.total IS NULL;

# 21) PostgresSQL Window Functions

Note that SUM, MIN, COUNT,MIN,MAX can all be used.

PostgreSQL’s documentation does an excellent job of [**introducing the concept of Window Functions**](https://www.postgresql.org/docs/9.1/static/tutorial-window.html): a window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Through introducing window functions, we have also introduced two statements that you may not be familiar with: **OVER**and **PARTITION BY**. These are key to window functions. Not every window function uses **PARTITION BY**; we can also use **ORDER BY** or no statement at all depending on the query we want to run. You will practice using these clauses in the upcoming quizzes. If you want more details right now, [**this resource**](https://blog.sqlauthority.com/2015/11/04/sql-server-what-is-the-over-clause-notes-from-the-field-101/) from Pinal Dave is helpful.

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Here is an example that shows how to compare each employee's salary with the average salary in his or her department:

SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;

depname | empno | salary | avg

-----------+-------+--------+-----------------------

develop | 11 | 5200 | 5020.0000000000000000

develop | 7 | 4200 | 5020.0000000000000000

develop | 9 | 4500 | 5020.0000000000000000

develop | 8 | 6000 | 5020.0000000000000000

develop | 10 | 5200 | 5020.0000000000000000

personnel | 5 | 3500 | 3700.0000000000000000

personnel | 2 | 3900 | 3700.0000000000000000

sales | 3 | 4800 | 4866.6666666666666667

sales | 1 | 5000 | 4866.6666666666666667

sales | 4 | 4800 | 4866.6666666666666667

(10 rows)

The first three output columns come directly from the table empsalary, and there is one output row for each row in the table. The fourth column represents an average taken across all the table rows that have the same depname value as the current row. (This actually is the same function as the regular avg aggregate function, but the OVER clause causes it to be treated as a window function and computed across an appropriate set of rows.)

A window function call always contains an OVER clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a regular function or aggregate function. The OVER clause determines exactly how the rows of the query are split up for processing by the window function. The PARTITION BY list within OVER specifies dividing the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

You can also control the order in which rows are processed by window functions using ORDER BY within OVER. (The window ORDER BY does not even have to match the order in which the rows are output.) Here is an example:

SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary;

depname | empno | salary | rank

-----------+-------+--------+------

develop | 8 | 6000 | 1

develop | 10 | 5200 | 2

develop | 11 | 5200 | 2

develop | 9 | 4500 | 4

develop | 7 | 4200 | 5

personnel | 2 | 3900 | 1

personnel | 5 | 3500 | 2

sales | 1 | 5000 | 1

sales | 4 | 4800 | 2

sales | 3 | 4800 | 2

(10 rows)

As shown here, the rank function produces a numerical rank within the current row's partition for each distinct ORDER BY value, in the order defined by the ORDER BY clause. rank needs no explicit parameter, because its behavior is entirely determined by the OVER clause.

The rows considered by a window function are those of the "virtual table" produced by the query's FROM clause as filtered by its WHERE, GROUP BY, and HAVING clauses if any. For example, a row removed because it does not meet the WHERE condition is not seen by any window function. A query can contain multiple window functions that slice up the data in different ways by means of different OVER clauses, but they all act on the same collection of rows defined by this virtual table.

We already saw that ORDER BY can be omitted if the ordering of rows is not important. It is also possible to omit PARTITION BY, in which case there is just one partition containing all the rows.

There is another important concept associated with window functions: for each row, there is a set of rows within its partition called its *window frame*. Many (but not all) window functions act only on the rows of the window frame, rather than of the whole partition. By default, if ORDER BY is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the ORDER BY clause. When ORDER BY is omitted the default frame consists of all rows in the partition. [[1]](https://www.postgresql.org/docs/9.1/static/tutorial-window.html" \l "FTN.AEN1050) Here is an example using sum:

SELECT salary, sum(salary) OVER () FROM empsalary;

salary | sum

--------+-------

5200 | 47100

5000 | 47100

3500 | 47100

4800 | 47100

3900 | 47100

4200 | 47100

4500 | 47100

4800 | 47100

6000 | 47100

5200 | 47100

(10 rows)

Above, since there is no ORDER BY in the OVER clause, the window frame is the same as the partition, which for lack of PARTITION BY is the whole table; in other words each sum is taken over the whole table and so we get the same result for each output row. But if we add an ORDER BY clause, we get very different results:

SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;

salary | sum

--------+-------

3500 | 3500

3900 | 7400

4200 | 11600

4500 | 16100

4800 | 25700

4800 | 25700

5000 | 30700

5200 | 41100

5200 | 41100

6000 | 47100

(10 rows)

Here the sum is taken from the first (lowest) salary up through the current one, including any duplicates of the current one (notice the results for the duplicated salaries).

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically execute after the processing of those clauses. Also, window functions execute after regular aggregate functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

If there is a need to filter or group rows after the window calculations are performed, you can use a sub-select. For example:

SELECT depname, empno, salary, enroll\_date

FROM

(SELECT depname, empno, salary, enroll\_date,

rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos

FROM empsalary

) AS ss

WHERE pos < 3;

The above query only shows the rows from the inner query having rank less than 3.

When a query involves multiple window functions, it is possible to write out each one with a separate OVER clause, but this is duplicative and error-prone if the same windowing behavior is wanted for several functions. Instead, each windowing behavior can be named in a WINDOW clause and then referenced in OVER. For example:

SELECT sum(salary) OVER w, avg(salary) OVER w

FROM empsalary

WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);

You can use dense\_rank if you do not want that the rank integers repeat, it will rank in numeric order not repeating the integers, but will assign the same rank if the ranks is equal;p:

SELECT o.id,o.account\_id,o.total, dense\_rank() OVER (PARTITION BY o.account\_id ORDER BY o.total DESC)

FROM orders o;

## WINDOW alisas example:

SELECT id,

account\_id,

DATE\_TRUNC('year',occurred\_at) AS year,

DENSE\_RANK() OVER account\_year\_window AS dense\_rank,

total\_amt\_usd,

SUM(total\_amt\_usd) OVER account\_year\_window AS sum\_total\_amt\_usd,

COUNT(total\_amt\_usd) OVER account\_year\_window AS count\_total\_amt\_usd,

AVG(total\_amt\_usd) OVER account\_year\_window AS avg\_total\_amt\_usd,

MIN(total\_amt\_usd) OVER account\_year\_window AS min\_total\_amt\_usd,

MAX(total\_amt\_usd) OVER account\_year\_window AS max\_total\_amt\_usd

FROM orders

WINDOW account\_year\_window AS (PARTITION BY account\_id ORDER BY DATE\_TRUNC('year',occurred\_at));

WINDOWS LEAD and LAG with example.

Lead finds the next row, while lag finds the previous row. After defining aliaises, you can perform mathematical operations.

SELECT account\_id,

standard\_sum AS total\_amt\_usd,

LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag,

LEAD(standard\_sum) OVER (ORDER BY standard\_sum) AS lead,

standard\_sum - LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag\_difference,

LEAD(standard\_sum) OVER (ORDER BY standard\_sum) - standard\_sum AS lead\_difference

FROM (

SELECT account\_id,

SUM(total\_amt\_usd) AS standard\_sum

FROM orders

GROUP BY 1

) sub

## NTILES

You can use window functions to identify what percentile (or quartile, or any other subdivision) a given row falls into. The syntax is NTILE(\*# of buckets\*). In this case, ORDER BY determines which column to use to determine the quartiles (or whatever number of ‘tiles you specify).

Note – if there are less bukets in the actual data, then the # of buckets will equal the actual date e.g. buckets(100) but only 28 buckets – then the NTILE result will be ordered by 28 buckets **SELECT** **id**,

account\_id,

occurred\_at,

total\_amt\_usd,

NTILE(100) **OVER** (**PARTITION** **BY** account\_id **ORDER** **BY** total\_amt\_usd) **AS** total\_percentile

**FROM** orders

**ORDER** **BY** account\_id **DESC**

Expert Tip

In cases with relatively few rows in a window, the NTILE function doesn’t calculate exactly as you might expect. For example, If you only had two records and you were measuring percentiles, you’d expect one record to define the 1st percentile, and the other record to define the 100th percentile. Using the NTILE function, what you’d actually see is one record in the 1st percentile, and one in the 2nd percentile. If you’re working with very small windows, keep this in mind and consider using quartiles or similarly small bands.

You can check out a complete list of window functions in Postgres (the syntax Mode uses) in the [**Postgres documentation**](https://www.postgresql.org/docs/8.4/static/functions-window.html).

Example:

SELECT account\_id,occurred\_at,standard\_qty,

NTILE(4) OVER (PARTITION BY account\_id ORDER BY standard\_qty) AS standard\_quartile

FROM orders

# 21) Performance considerations

One way to make a query run faster is to reduce the number of calculations that need to be performed. Some of the high-level things that will affect the number of calculations a given query will make include:

* Table size
* Joins
* Aggregations

Query runtime is also dependent on some things that you can’t really control related to the database itself:

* Other users running queries concurrently on the database
* Database software and optimization (e.g. Postgres is optimized differently than Redshift)
* EXPLAIN can be put before the SQL statement to get an idea about the performance implications of your SQL statement
* Note the COUNT DISTINCT is very performance expensive because the database has to verify the uniqueness of the rows

Comments: Table size and aggregation affect performance. The Udacity lecturer pointed out that sub-queries can be used to filter data that is going into test tables. A key thing to remember is that agregations are performed on the whole table – this means that a limit will not save you. The main thing to remember is to use subqueries to make smaller test tables so that you can work faster. When the query logic has been verified, then you can try it on the whole table. Large tables could take many hours to execute some queries.

For with statements, you have to look at the data base being used. For PostgreSQL with statements will not increase performance – subqueries should be used.