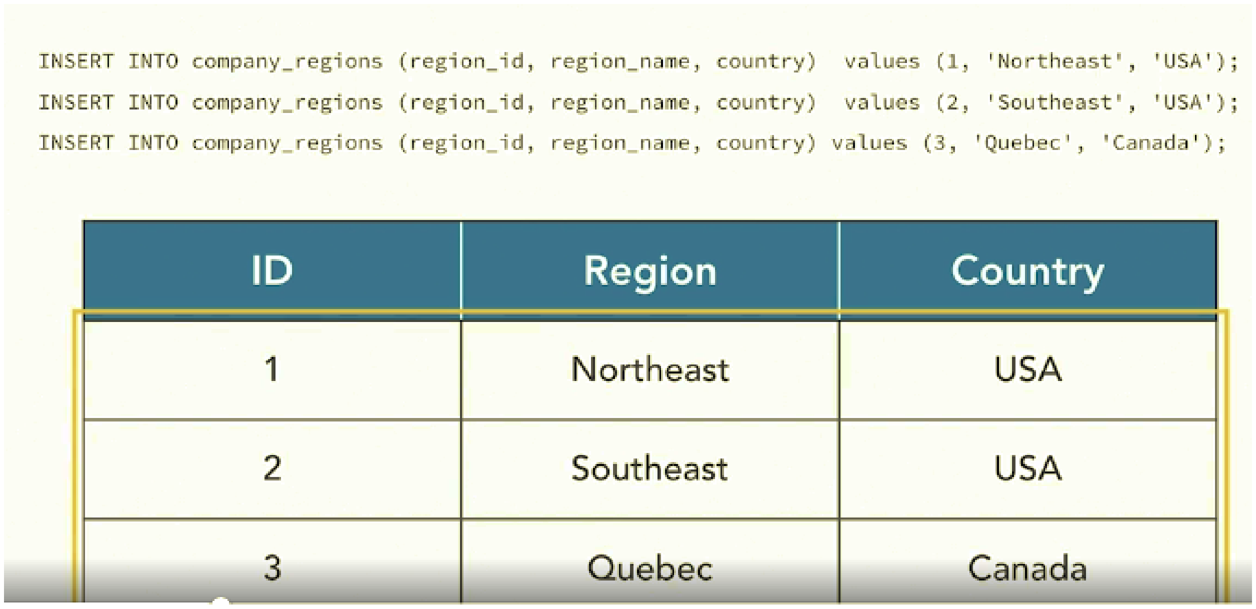
Foundations of SQL for Data Sci

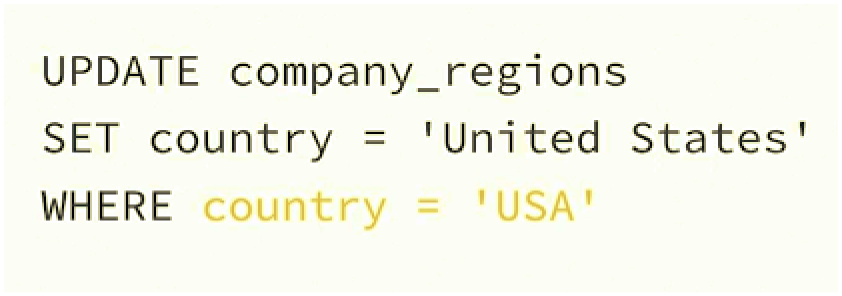
INSERT



```
INSERT INTO company_regions (region_id, region_name, country)  values (1, 'Northeast', 'USA');
INSERT INTO company_regions (region_id, region_name, country)  values (2, 'Southeast', 'USA');
INSERT INTO company_regions (region_id, region_name, country) values (3, 'Quebec', 'Canada');
```

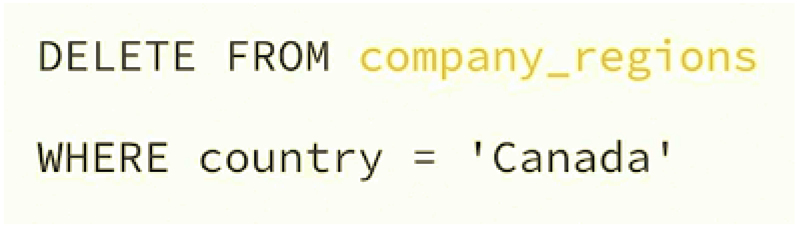
ID	Region	Country
1	Northeast	USA
2	Southeast	USA
3	Quebec	Canada

UPDATE



```
UPDATE company_regions
SET country = 'United States'
WHERE country = 'USA'
```

DELETE

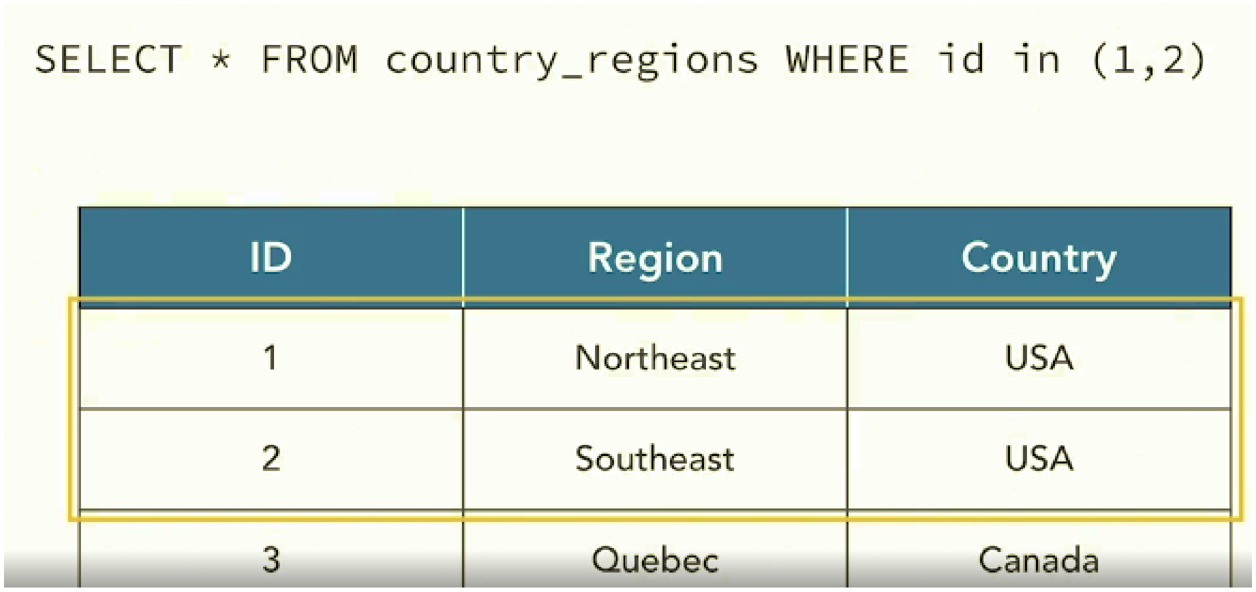


```
DELETE FROM company_regions

WHERE country = 'Canada'
```

SELECT

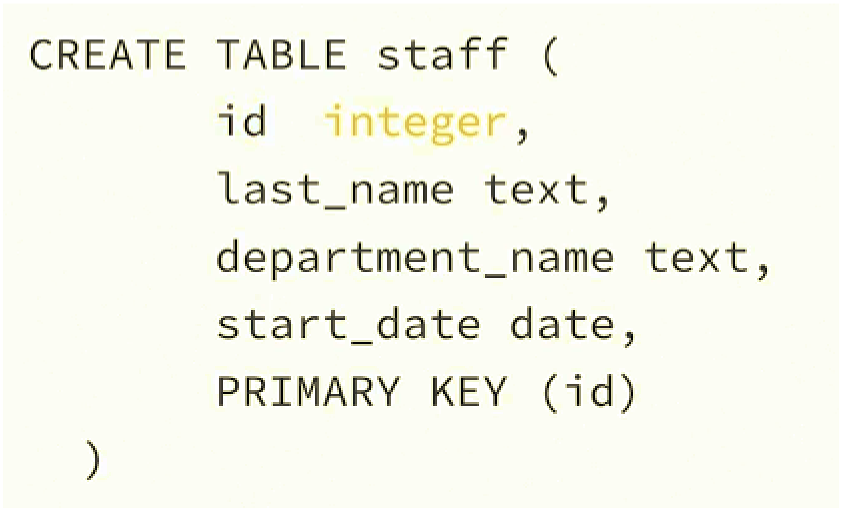
- One or more table name s
- A list of columns to retrieve
- One or more joins of tables
- WHERE clauses
- Aggregate functions



```
SELECT * FROM country_regions WHERE id in (1,2)
```

ID	Region	Country
1	Northeast	USA
2	Southeast	USA
3	Quebec	Canada

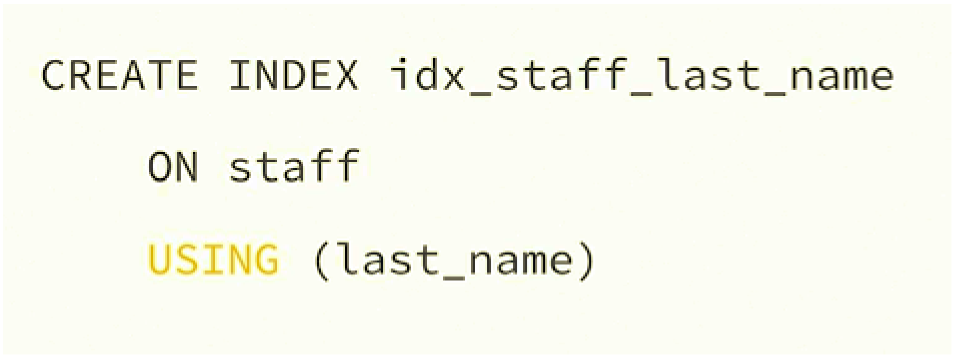
<u>TABLES</u>



```
CREATE TABLE staff (
       id  integer,
       last_name text,
       department_name text,
       start_date date,
       PRIMARY KEY (id)
  )
```

INDEXES

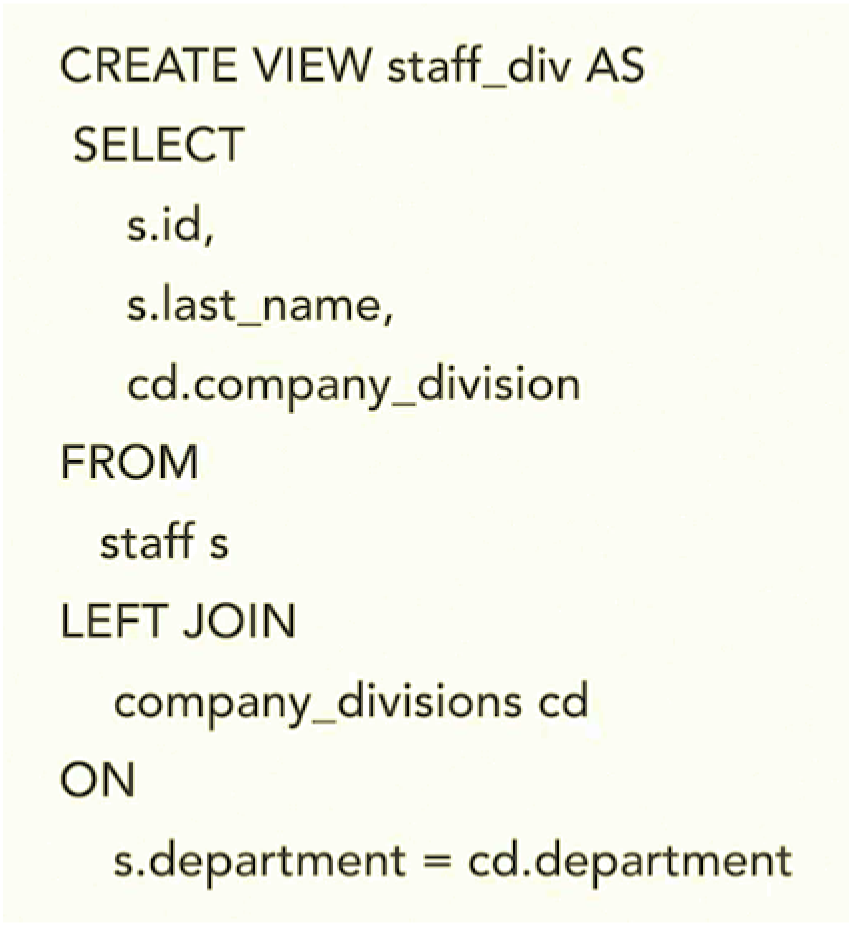
- Used to improve performance
- Create an index on the staff table and use the last name.



```
CREATE INDEX idx_staff_last_name
    ON staff
    USING (last_name)
```

VIEWS

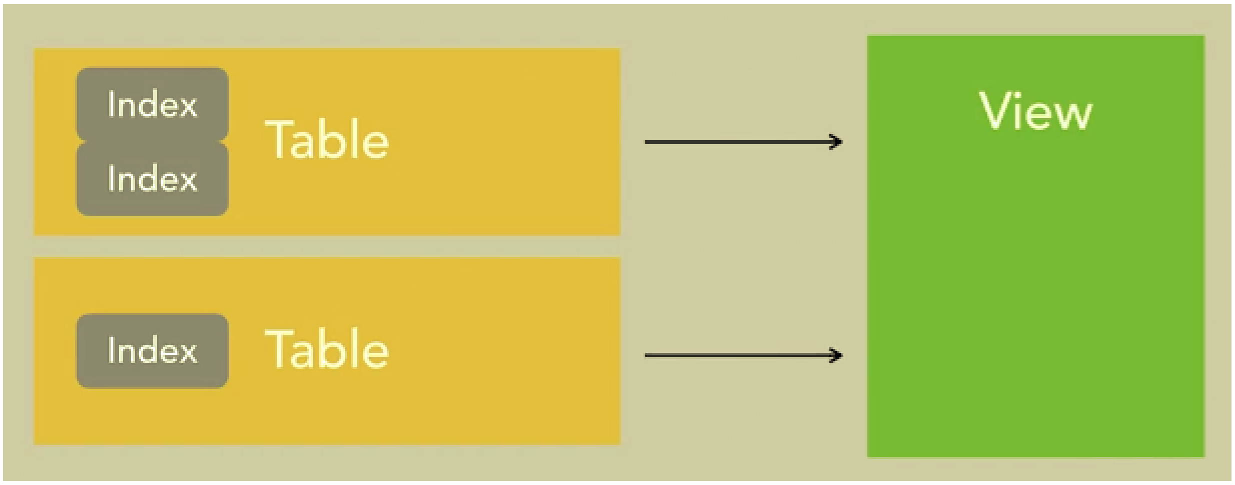
- Allow us to create a relation or projection.
- We are not creating new tables, we are simply collecting data from one or more tables and presenting it as a table, and manipulating accordingly
- Giving the view a name as staff_div. Specify the select statement columns from staff, left join so every row in the staff table will be included. This will provide a view which shows staff id, staff last name, and then company division from company divisions table.



```
CREATE VIEW staff_div AS
 SELECT
    s.id,
    s.last_name,
    cd.company_division
FROM
   staff s
LEFT JOIN
    company_divisions cd
ON
    s.department = cd.department
```

SCHEMAS

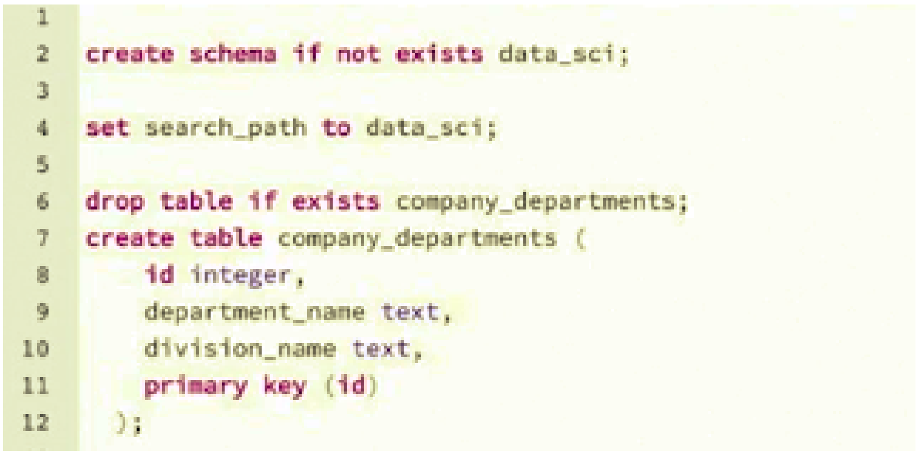
- Schemas are collections or grouping structures.
- Way or organizing tables and views.



BASIC STATISTICS WITH SQL

LOADING DATA

- In order to create schema and tables to work with we navigate to the schemas tab and select the query tool. From this point we upload our given file which allows us to create tables with data.
- The if not exists statement allows us to rerun the script multiple times. It should be pointed out that if we use dashes in table names they must be surrounded by quotes. Therefore underscores are recommended.
- The search path says if we do not specify a schema work with the data science schema.
- PostGres supports Var Chart but it can potentially create restraint issues.
- The drop table if exists once again allows us to run the script multiple times.
- After executing the script we can refresh the schema tab, and scroll down to tables to confirm their existence.



```

create schema if not exists data_sci;

set search_path to data_sci;

drop table if exists company_departments;
create table company_departments (
    id integer,
    department_name text,
    division_name text,
    primary key (id)
  );
```

BASIC AGGREGATE FUNCTIONS

- Tools —-. Query Tool to enter command
- Select the first 10 records from data_sci employees table
- Select everything from data_sci employees table where region id is equal to 2
- Select count of records from data sci employee table
- Return count of records, minimum value of salary column, and maximum value of salary column
- Return count of records, minimum value of salary column, and maximum value of salary column where region_id is equal to 2.
- Select count of rows, minimum, and maximum value of id from employees table.

```
select *
from
data_sci.employees
limit 10
```

```
select *
from
data_sci.employees
where region_id = 2
```

```
select
count(*)
from
data_sci.employees
```

```
select
count(*), min(salary), max(salary)
from
data_sci.employees
```

```
select
count(*), min(salary), max(salary)
from
data_sci.employees
where
region_id = 2
```

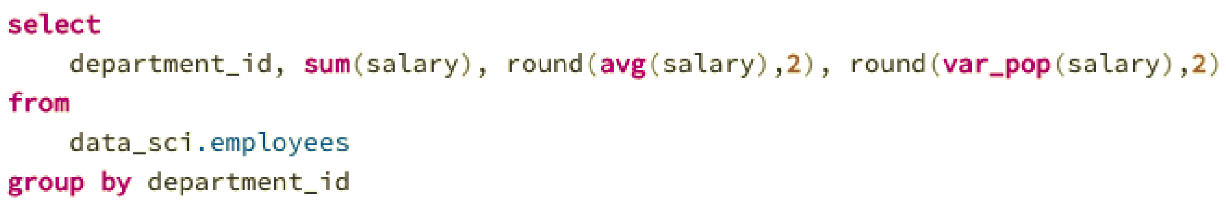
```
select
count(*), min(id), max(id)
from
data_sci.employees
```

STATISTICAL AGGREGATE FUNCTIONS

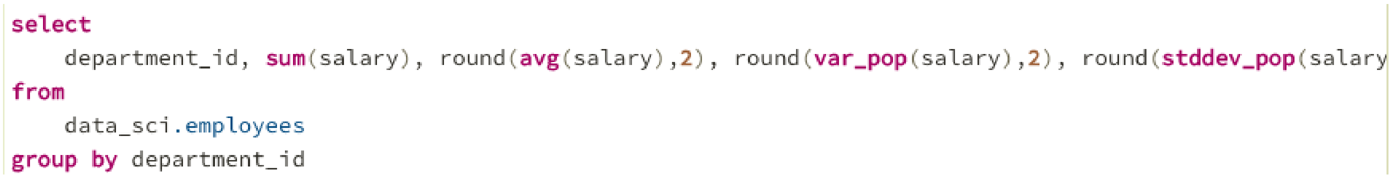
- Find the total salary across all departments in the employee table using groupby.
- Select department id as a column, select average salary as a column round to 2 decimal points, and sum the salary column while grouping by department id.
- Select all the same columns, and find the variance of the salary distribution in their respective groups. (The variation's square root is the standard deviation)
- Find the standard deviation of the salary column while grouped by department id. Stddev gives us a number that is easy to work with within a given range. We know within one standard deviation of the mean that about 60-66 percent of the salaries are within that given range. This works best with normal distribution. Without the presence of this standard deviation and variance lose some of their effectiveness.

```
select
    department_id, sum(salary)
from
    data_sci.employees
group by department_id
```

```
select
    department_id, sum(salary), round(avg(salary),2)
from
    data_sci.employees
group by department_id
```



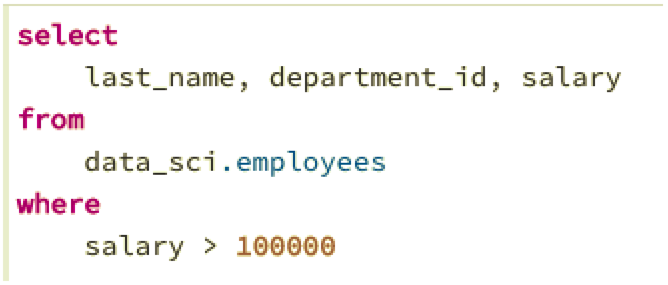
```
select
    department_id, sum(salary), round(avg(salary),2), round(var_pop(salary),2)
from
    data_sci.employees
group by department_id
```



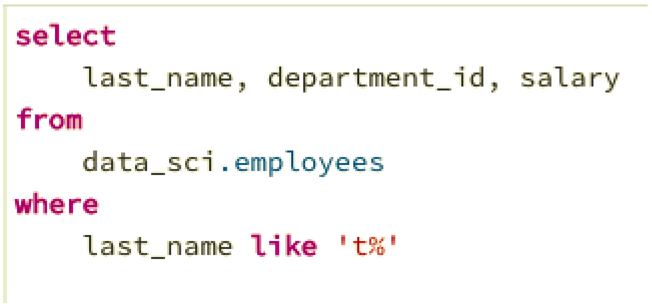
```
select
    department_id, sum(salary), round(avg(salary),2), round(var_pop(salary),2), round(stddev_pop(salary
from
    data_sci.employees
group by department_id
```

GROUPING AND FILTERING DATA

- Select the columns last name, department_id, and salary from the data science employee table where the salary is greater than 100000
- Filter on character strings, check for last names that start with the letter T.
  It is important to mention that the operator was changed from an equal sign to like statement.
- One step further for last names that begin with b and end with d & salary greater than 100000
- By changing the and statement to an or statement only one of the statements need to be true in order to be responded to in the query.
- For instances in specific departments where employees salaries are greater than 100000 return the sum by department id.



```
select
    last_name, department_id, salary
from
    data_sci.employees
where
    salary > 100000
```



```
select
    last_name, department_id, salary
from
    data_sci.employees
where
    last_name like 't%'
```

```
select
    last_name, department_id, salary
from
    data_sci.employees
where
    last_name like 't%r'
and
    salary > 100000
```

```
select
    last_name, department_id, salary
from
    data_sci.employees
where
    last_name like 'b%'
or
    salary > 100000
```

```
select
    sum(salary)
from
    data_sci.employees
where
    salary > 100000
group by
    department_id
```

JOINING AND FILTERING DATA

- Join employees table and company regions table, joining on company regions id (PK) & the employees region id. (FK)
  This will return all columns from employees table, and all columns from company regions
- We can create an alias for tables - show below
- Eliminate the extra id column that was in the company_regions table by only selecting the other two rows.
- Add a where clause where we narrow down the country name to canada. Keep in mind that these queries are case sensitive.

- When using code over and over again if someone goes into the database and adds a new column it can be detrimental because the code selects *. Therefore be explicit when you know you are going to use a code over and over again.
- Example of an explicitly stating all the columns. This can be useful for someone in the business domain as we have taken out all relevant database ids.

```
select
    *
from
    data_sci.employees
join
    data_sci.company_regions
on
    employees.region_id = company_regions.id
```

```
select
    *
from
    data_sci.employees e
join
    data_sci.company_regions cr
on
    e.region_id = cr.id
```

```
select
    e.*, cr.region_name, cr.country_name
from
    data_sci.employees e
join
    data_sci.company_regions cr
on
    e.region_id = cr.id
```

```
select
    e.*, cr.region_name, cr.country_name
from
    data_sci.employees e
join
    data_sci.company_regions cr
on
    e.region_id = cr.id
where
    cr.country_name = 'canada'
```

```
select
    e.last_name,
    e.email,
    e.start_date,
    e.salary,
    e.job_title,
    cr.region_name,
    cr.country_name
from
    data_sci.employees e
join
    data_sci.company_regions cr
on
    e.region_id = cr.id
where
    cr.country_name = 'canada'
```

PRACTICE CHALLENGE

- Write a select query to return last name, email, department name, for employees with salaries greater than 120,000.

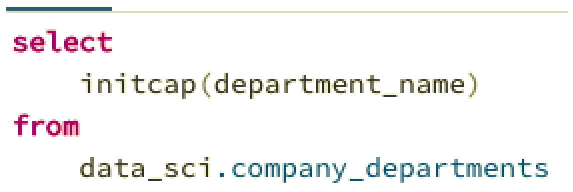
```
select
    employees.last_name,
    employees.email,
    employees.salary,
    company_departments.department_name
from
    data_sci.employees, data_sci.company_departments
where
    salary > 120000
```

DATA MUNGING WITH SQL

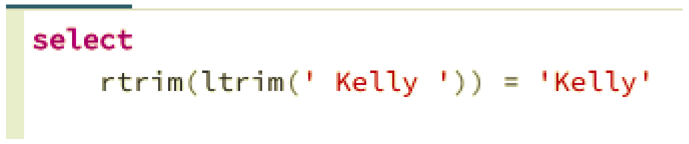
REFORMAT CHARACTER DATA

- Select department name and reformat all records as uppercase.
- To only capitalize the first letter we use the initcap, to convert back to lower just add lower keyword
- Say there are extra spaces at the beginning or end when pulling data from another system. One thing we can do is trim the extra space - ltrim function.
- Provides a combination of rtrim and ltrim to take off whitespace.
- Combine an employee's title and last name put together into one column. We can do this through the pipe operators.
- Say there is an instance where there are null values in columns, well then we would use concat instead because it will still return values.
- Concat with separator allows us to place the separator at the beginning and then add variables.

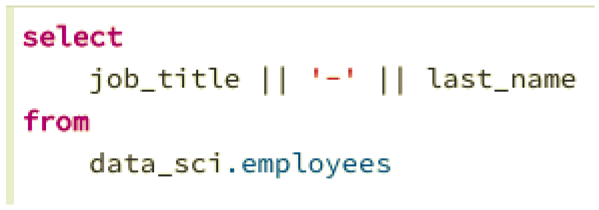
```
select
    upper(department_name)
from
    data_sci.company_departments
```



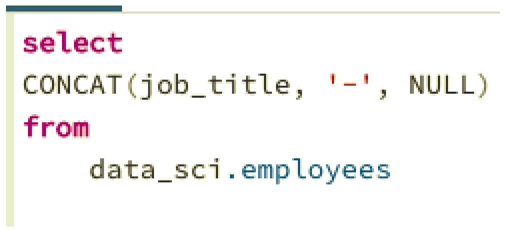
```
select
    initcap(department_name)
from
    data_sci.company_departments
```



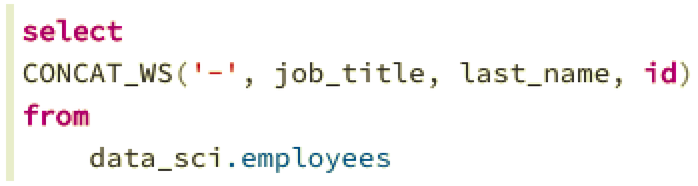
```
select
    rtrim(ltrim(' Kelly ')) = 'Kelly'
```



```
select
    job_title || '-' || last_name
from
    data_sci.employees
```



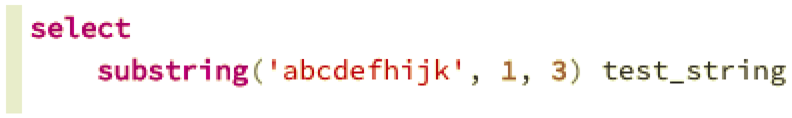
```
select
CONCAT(job_title, '-', NULL)
from
    data_sci.employees
```



```
select
CONCAT_WS('-', job_title, last_name, id)
from
    data_sci.employees
```

<u>EXTRACT STRINGS FROM CHARACTER DATA</u>

- Say we wanted to extract a substring…keep in mind first character is indexed as 1
- Start at a given index position and extract the rest of the string
- Search for all employees in the table that have the word assistant in their job title. We utilize the like statement with the wildcard percent sign before and after to get records back that have words before and after assistant. This is a full table scan that does not work with indexing so the query may take longer than usual.
- Create a new boolean expression column for job titles that have assistants within their job title.



```
select
    substring('abcdefhijk', 1, 3) test_string
```

```
select
    substring('abcdefhijk' from 3) test_string
```

```
select
    *
from
    data_sci.employees
where
    job_title like '%assistant%'
```

```
select
    job_title, salary, (job_title like '%assistant%') is_assistant
from
    data_sci.employees
```

FILTER WITH REGULAR EXPRESSIONS

- Select unique or distinct job titles
- Select distinct job titles with names similar to vp…, and web….
  This same expression can be queried through an or statement as well.
  Given the whitespace is within the pipe string there can not be any spaces.
- If we wanted to do the same query but search for a job title that begins with vp with the word beginning with a.
- Vp followed by an a or an m.

```
select distinct
    job_title
from
    data_sci.employees
```

```
select distinct
    job_title
from
    data_sci.employees
where
    job_title similar to '(vp%|web%)'
```

```
select distinct
    job_title
from
    data_sci.employees
where
    job_title similar to 'vp a%'
```

```
select distinct
    job_title
from
    data_sci.employees
where
    job_title similar to 'vp (a|m)%'
```

REFORMAT NUMERIC DATA

- To completely remove decimal points from a query we can use trunc. Trunc can also take an argument for the cutoff point with decimals.
- Automatically round up to the nearest whole number with the ceil operator.

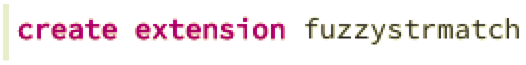
```
select
    trunc(avg(salary))
from
    data_sci.employees
```

```
select
    ceil(avg(salary))
from
    data_sci.employees
```

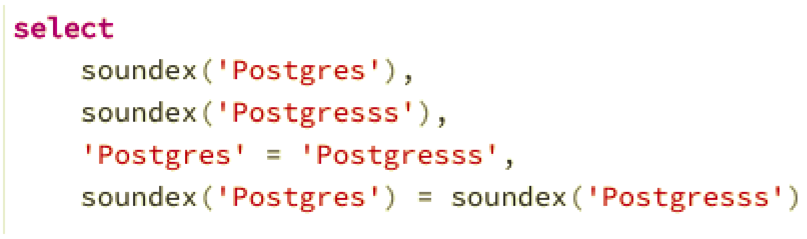
USE SOUNDEX WITH MISSPELLED TEXT

- Must first install extension
- Soundex function returns a code for a piece of text. For example, the soundex of postgres returns p232.
- The soundex function can allow us to match up on terms that may potentially be misspelled.
- Show how close two strings are in relation to each other when sounded out through soundex and difference

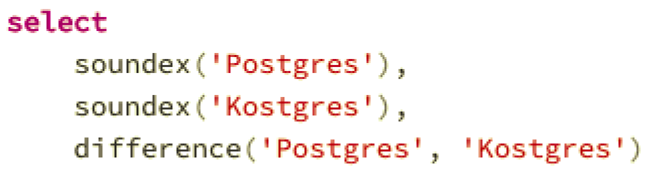
- The Levenshtein function tells us the number of operations we must perform in order to get it to match the other string. Between Postgres, and Lostgres it would be 1.
- For example we may have long strings, and we can bring in Levenshtein to determine the error point and what is tolerable.



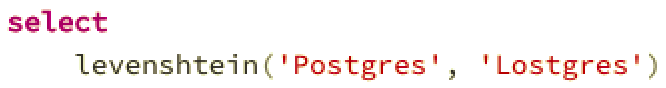
```
create extension fuzzystrmatch
```



```
select
    soundex('Postgres'),
    soundex('Postgresss'),
    'Postgres' = 'Postgresss',
    soundex('Postgres') = soundex('Postgresss')
```



```
select
    soundex('Postgres'),
    soundex('Kostgres'),
    difference('Postgres', 'Kostgres')
```



```
select
    levenshtein('Postgres', 'Lostgres')
```

CHALLENGE: PREPARE A DATA SET FOR ANALYSIS

- What functions could you use to reformat a number from having eight decimal places to two decimal places?
  Can use round, truncate, by specifying 2 as the second parameter.
- What operator would you use to filter using regular expression
  Similar to
- What function would you use to measure the difference in characters between two strings?
  Levenshtein

FILTERING & AGGREGATION

USE THE HAVING CLAUSE TO FIND SUBGROUPS

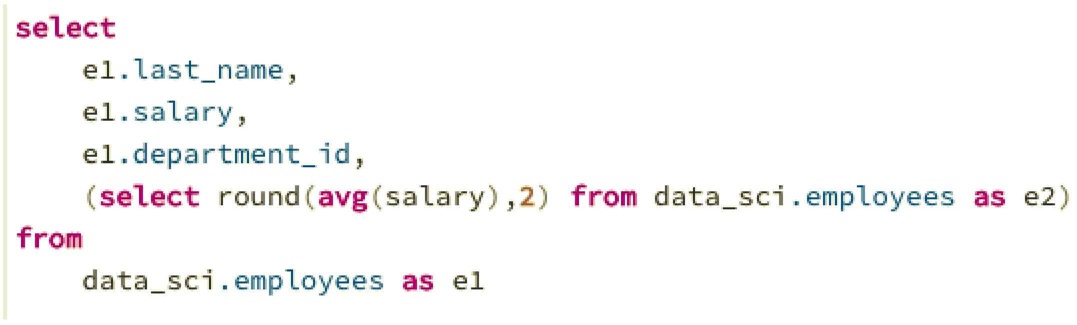
- Between employee table, and company_departments table select the department_name column from company departments and count up employees. The count of employees are displayed by department name group by. The count of everything is displayed in descending order.
- Where clauses are not allowed in aggregate functions, for where clauses determine what rows go into a particular result set. The solution to this is the having clause. It behaves like where and filters out rows in the result set that include the aggregate results.
- We have 5 departments each with a count over 50. This allows to filter of the aggregate function output.

```
select
    cd.department_name, count(*)
from
    data_sci.employees as e
join
    data_sci.company_departments as cd
on
    e.department_id = cd.id
group by
    cd.department_name
order by
    count(*) desc
```

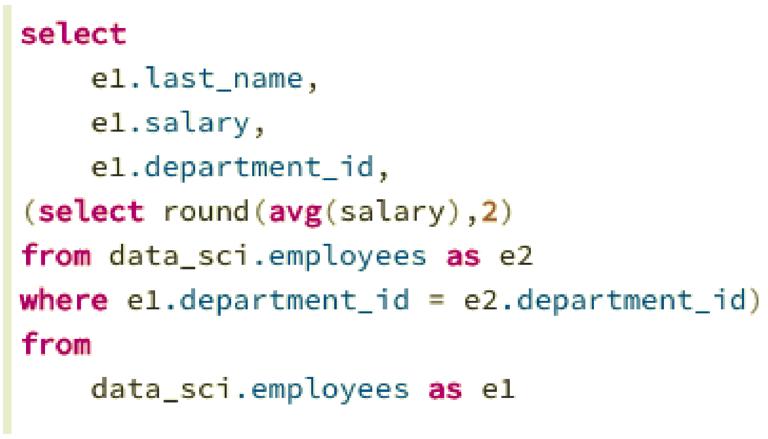
```
select
    cd.department_name, count(*)
from
    data_sci.employees as e
join
    data_sci.company_departments as cd
on
    e.department_id = cd.id
group by
    cd.department_name
having
    count(*) > 50
order by
    cd.department_name
```

SUBQUERIES OF COLUMN VALUES

- Select last name, salary, department id from data sci employees as e1, then create a subquery that takes the salary from the original statement and creates an average salary column.
  Because we have data_sci.employees mentioned twice we need to distinguish from the outer data science employees, and the subquery reference. The solution is e1 & e2 when referencing.
- We implement a where clause in order to put the average salary by department. We get the department name from the outer query. We use the e1 table to get department_id, and make sure to match up on rows from the employees table where the department_ids match by row before providing average salary.



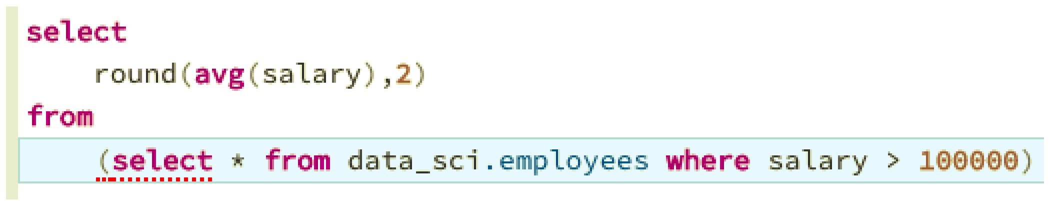
```
select
    e1.last_name,
    e1.salary,
    e1.department_id,
    (select round(avg(salary),2) from data_sci.employees as e2)
from
    data_sci.employees as e1
```



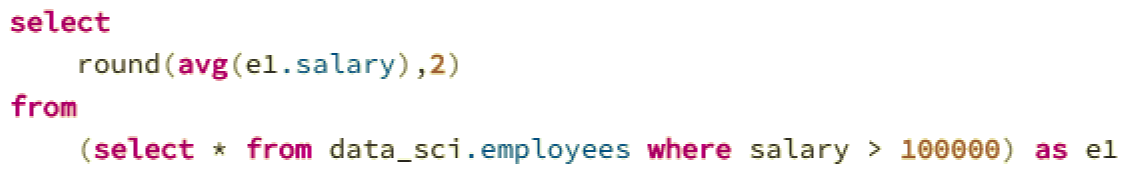
```
select
    e1.last_name,
    e1.salary,
    e1.department_id,
(select round(avg(salary),2)
from data_sci.employees as e2
where e1.department_id = e2.department_id)
from
    data_sci.employees as e1
```

<u>SUBQUERIES IN FROM CLAUSES</u>

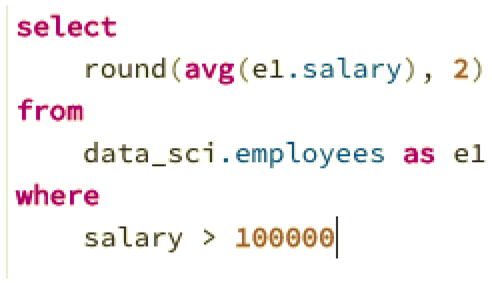
- The subquery below will throw an error because there is no alias. The select clause has no name in it, and it is an anonymous clause. Therefore we give it a label as e1.
- The alternative is the third box, but the reason why we would use a subquery in a where clause is because we have some kind of complex logic, and it is more readable.



```
select
    round(avg(salary),2)
from
    (select * from data_sci.employees where salary > 100000)
```



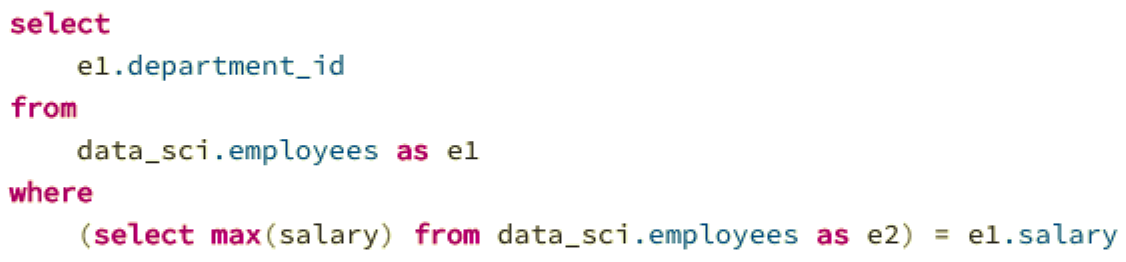
```
select
    round(avg(e1.salary),2)
from
    (select * from data_sci.employees where salary > 100000) as e1
```



```
select
    round(avg(e1.salary), 2)
from
    data_sci.employees as e1
where
    salary > 100000
```

<u>SUBQUERIES IN WHERE CLAUSES</u>

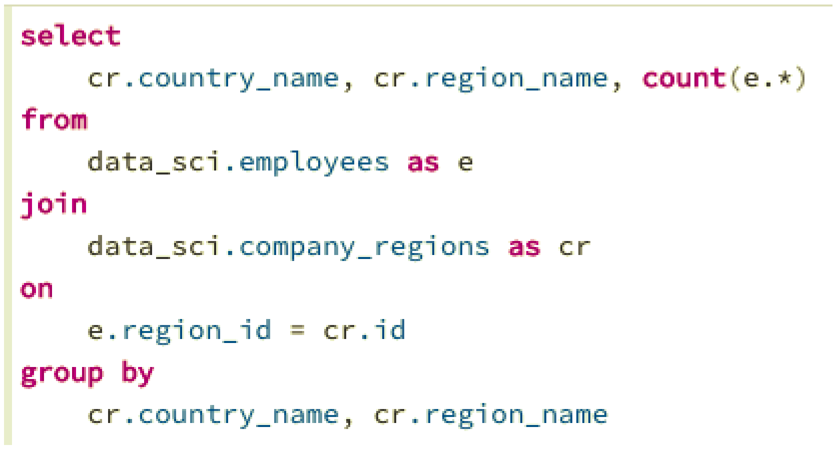
- Find the department that has the employee with the maximum salary
  When we use subqueries in where clauses we have to make sure we have a boolean expression. For example, we are selecting the maximum query, and then looking at the primary e1 comparing each salary to the max. When the salary matches the max it is selected and returned.



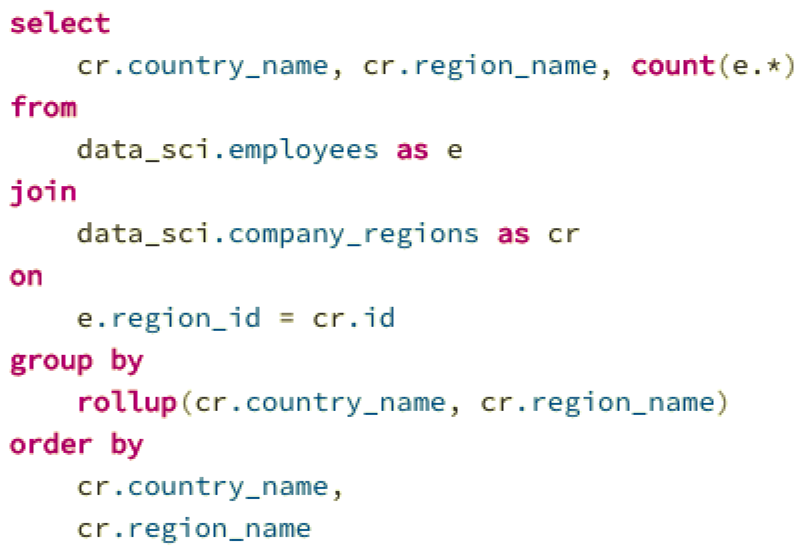
```
select
    e1.department_id
from
    data_sci.employees as e1
where
    (select max(salary) from data_sci.employees as e2) = e1.salary
```

USE ROLLUP TO CREATE SUBTOTALS

- Rollup is used when working with aggregates to find subgroups
- Given the first query we have country_name, region_name, total number of employees in each region, joined on ids.
- Say we also wanted to know what the total count is for the country specifically. This can be done through the rollup operator
- The rollup function is paired with the rollup to total the number of employees for Canada, USA, and then both. This allows us to identify subgroup aggregates according to the structure of the hierarchy. For example, with country and region it is just two levels.



```
select
    cr.country_name, cr.region_name, count(e.*)
from
    data_sci.employees as e
join
    data_sci.company_regions as cr
on
    e.region_id = cr.id
group by
    cr.country_name, cr.region_name
```



```
select
    cr.country_name, cr.region_name, count(e.*)
from
    data_sci.employees as e
join
    data_sci.company_regions as cr
on
    e.region_id = cr.id
group by
    rollup(cr.country_name, cr.region_name)
order by
    cr.country_name,
    cr.region_name
```

USE CUBE TO TOTAL ACROSS DIMENSIONS

- We select country_name, region_name, and department_name. We count up the employees, join company_regions on employees region_id and employees region_id. Then that frame is further joined to company_departments on company departments id. After the joins the numbers are grouped by the columns.
- When we input cube in the group by statement it groups each unique combination in every column. For example, every department_name matched with Canada. (grouping by 2 dimensions) Then we have just Canada total , and Canada with Region Name, and then just departments. The cube builds all possible combinations.
- This is useful for data exploration

```
select
    cr.country_name,
    cr.region_name,
    cd.department_name,
    count(e.*)
from
    data_sci.employees as e
join
    data_sci.company_regions as cr
on
    e.region_id = cr.id
join
    data_sci.company_departments as cd
on
    e.department_id = cd.id
group by
    cr.country_name,
    cr.region_name,
    cd.department_name
order by
    cr.country_name,
    cr.region_name
```

```
select
    cr.country_name,
    cr.region_name,
    cd.department_name,
    count(e.*)
from
    data_sci.employees as e
join
    data_sci.company_regions as cr
on
    e.region_id = cr.id
join
    data_sci.company_departments as cd
on
    e.department_id = cd.id
group by
    cube(cr.country_name,
        cr.region_name,
        cd.department_name)
order by
    cr.country_name,
    cr.region_name
```

USE TOP N-QUERIES TO FIND TOP RESULTS

- Select everything from the data science employee table descend by salary rank and return the first 10 rows. (not standard SQL).
- Standard SQL version -

```
select
    *
from
    data_sci.employees
order by
    salary desc
limit 10
```

```
select
    *
from
    data_sci.employees
order by
    salary desc
fetch first 10 rows only
```

CHALLENGE: FILTER AND AGGREGATE A DATA SET

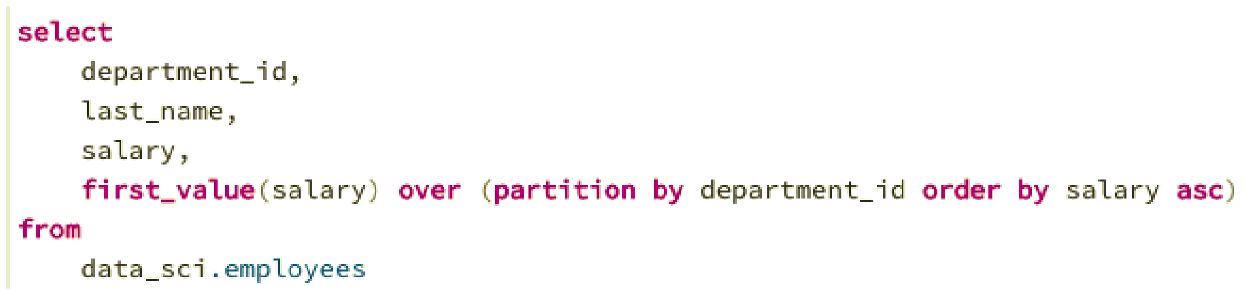
- Write a query to return the count of employees in departments where the total salary paid in that department is greater than $5,000,000

```
select
    department_id, sum(salary)
from
    data_sci.employees
group by
    department_id
having
    sum(salary) > 5000000
order by
    sum(salary) desc
```

WINDOW FUNCTIONS ARE ORDERED DATA

INTRODUCTION TO WINDOW FUNCTIONS

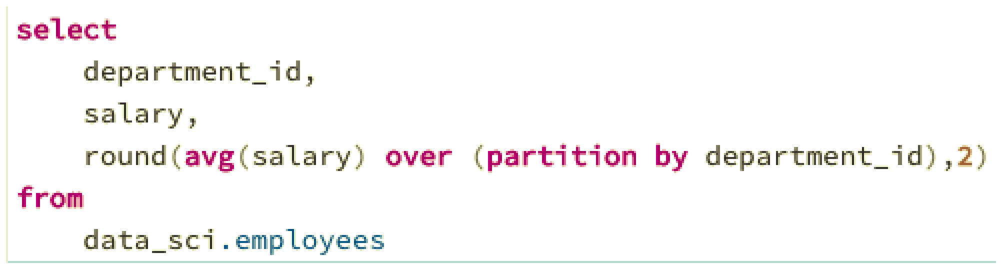
- Window is a metaphor for a window over a set of rows. The general idea is a window allows us to look at a small subgroup of rows. Then we can pick up a window and move it to another group and apply the same function.
- Group by departments to find who has the lowest salary in each department. One way to do that is by using the first_value in that group. Then we tell how we want to move the window through the partition statement. In this case we partition by department id.
- This returns the value of salaries sorted or partitioned on department_id in ascending order.



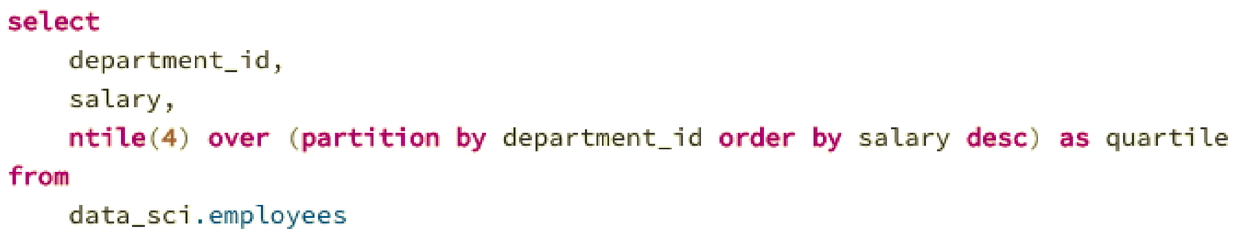
```
select
    department_id,
    last_name,
    salary,
    first_value(salary) over (partition by department_id order by salary asc)
from
    data_sci.employees
```

NTH VALUE AND NTILE

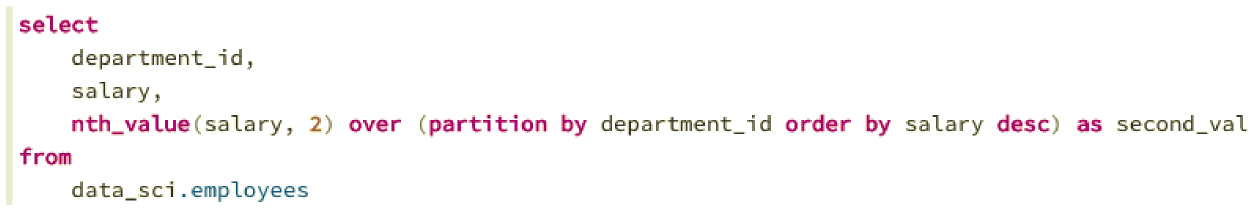
- We round the average salary by department. The round statement must encapture the partition piece.
- The query will break data up into quartiles based on salary number. The feedback is grouped or partitioned on department_id, and ordered by descending salary. Therefore this query will return grouped quartiles of salary based on department.
- Select department id and salary, then create additional column that displays the second value of salary while partitioning on department id, in salary descending order.



```
select
    department_id,
    salary,
    round(avg(salary) over (partition by department_id),2)
from
    data_sci.employees
```



```
select
    department_id,
    salary,
    ntile(4) over (partition by department_id order by salary desc) as quartile
from
    data_sci.employees
```



```
select
    department_id,
    salary,
    nth_value(salary, 2) over (partition by department_id order by salary desc) as second_val
from
    data_sci.employees
```

RANK, LEAD, AND LAG

- Partition by department, and give a ranking in descending order by salary. This provides an extra column with rankings.
- The second query would do the same thing but across the whole set. It is not useful in this sense because the index does the same thing.
- Lead is looking one ahead and in this case it grabs the salary for an index position that is one place lower. This number can be specified within the lead function.
- Lag will do the opposite as it will look back rows

```sql
select
    department_id,
    salary,
    last_name,
    rank() over (partition by department_id order by salary desc)
from
    data_sci.employees
```

```sql
select
    department_id,
    salary,
    last_name,
    rank() over (order by salary desc)
from
    data_sci.employees
```

```sql
select
    department_id,
    salary,
    last_name,
    lead(salary) over (partition by department_id order by salary desc)
from
    data_sci.employees
```

WIDTH_BUCKET and CUME_DIST

- The width_bucket function will create a new column and give a ranking like a histogram would.
- Below is an instance of casting in Postgres. We round the cumulative distribution number by converting from a double precision to numeric.

```
select
    department_id,
    last_name,
    salary,
    width_bucket(salary, 0, 150000, 6)
from
    data_sci.employees
order by width_bucket


select
    department_id,
    last_name,
    salary,
    round((cume_dist() over (order by salary desc))::numeric,3)
from
    data_sci.employees
```

CHALLENGE: SEGMENT A DATA SET USING WINDOW FUNCTIONS

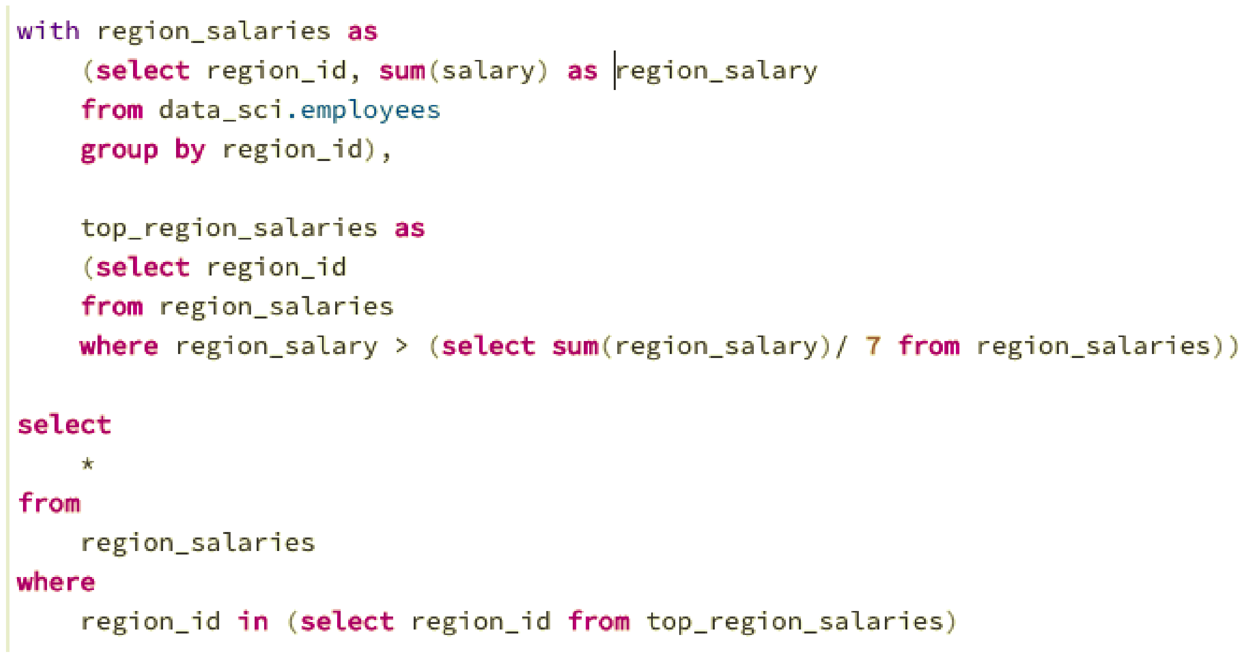
- Write a query to return department_id, last_name salary, and sum of all salaries in a department. (Do not use a subquery).
-

```
select
    department_id,
    last_name,
    salary,
    sum(salary) over (partition by department_id)
from
    data_sci.employees
```

COMMON TABLE EXPRESSIONS

INTRO TO CTES

- Common table expressions allow us to create temporary tables used only within a query
- Allows us to simplify complex queries.
- Allow us to work with recursive data structures, such as hierarchical data
- The with statement allows us to define region_salaries.
- Top region salaries is defined by selecting the region id from region salaries (the cte we just created) we select a subset where the region salary is greater than the average.



```
with region_salaries as
    (select region_id, sum(salary) as region_salary
    from data_sci.employees
    group by region_id),

    top_region_salaries as
    (select region_id
    from region_salaries
    where region_salary > (select sum(region_salary)/ 7 from region_salaries))

select
    *
from
    region_salaries
where
    region_id in (select region_id from top_region_salaries)
```

HIERARCHICAL TABLES

- The recursive statement allows us to query over hierarchical tables. When working with recursion we need a terminal select statement where we bottom out, and then we union them to results of the recursive select statement. Then reference the primary select.
- The org structure table is created. The parent of VP Sales, and VP Operations are CEO Office. The parent of Northwest Sales, and Northeast sales is VP sales, and lastly the parent of Infrastructure Operations and Management Operations is VP Operations.

```
drop table if exists data_sci.org_structure;
create table data_sci.org_structure (
    id integer,
    department_name text,
    parent_department_id integer);


insert into data_sci.org_structure values
  (1, 'CEO Office', null),
  (2, 'VP Sales', 1),
  (3, 'VP Operations', 1),
  (4, 'Northeast Sales',2),
  (5, 'Northwest Sales',2),
  (6, 'Infrastructure Operations', 3),
  (7, 'Management Operations', 3);
```

RECURSIVE COMMON TABLE EXPRESSIONS

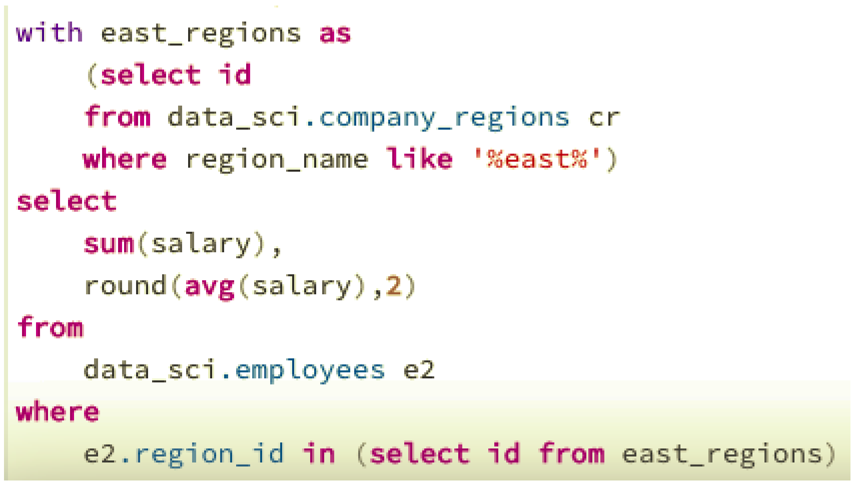
- We can use recursive CTEs to find paths through trees from a leaf node to a root.
- The first line defines the path with a recursive cte as report structure, and the second line selects the information we want from org_structure table where id = 7. (This is where the terminal select statement bottoms out.) Now we union that with results from the recursive select statement. (When it is implemented we use iteration - same as recursion)
  We select id, department_name, parent_department_id from data_sci org structure. Then to implement the incursion we must state a join on report structure on report structure parent_department_id. Then we call on the recursive function by selecting everything from report structure.

```
with recursive report_structure (id, department_name, parent_departmentid) as
(select id, department_name, parent_department_id
from data_sci.org_structure where id = 7
 union
 select os.id, os.department_name, os.parent_department_id
 from data_sci.org_structure as os
 join report_structure as rs
 on rs.parent_departmentid = os.id)
 select
    *
 from
    report_structure
```

CHALLENGE: REWRITE A COMPLEX QUERY TO USE CTES

- Create east_regions common table expression by selecting id from data_sci.company_region table where the region name has east in it.

- Select the sum of salary and round that number from data_sci employees table where the region id is also in the east_regions common table expression.



```
with east_regions as
    (select id
    from data_sci.company_regions cr
    where region_name like '%east%')
select
    sum(salary),
    round(avg(salary),2)
from
    data_sci.employees e2
where
    e2.region_id in (select id from east_regions)
```