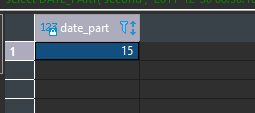
# Lecture 6

## 03/11/2021

Date part function

|  |
| --- |
| select DATE\_PART('second', '2011-12-30 08:56:10'::timestamp - '2011-12-30 08:54:55'::timestamp); |



Input:

* Varchar denoting the increment
* Two timestamp values between which the difference needs to be calculated

Note: In the example above the timestamp cast is being done by the ‘::’ operator

Output

An integer number of seconds giving the difference between the two timestamp values.

**Grain:** denotes the lowest denominator of the date; what a single row should represent in the dataset

Example: What is the daily trip count?

-grain/row: daily observation count

What date column to use to establish the grain?

Tpep\_pickup\_datetime → date only

|  |
| --- |
| select date\_trunc('day', tpep\_pickup\_datetime) from qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz" yttdg |

Input:

* Varchar value denoting the grain of your datetime
* Timestamp data type denoting the actual variable that needs to be truncated

Output: Timestamp data type truncated to the value that we established in the variable

Question: What is the daily trip count?

Sub Questions:

* What is the grain of the question?
  + tpep\_pickup\_datetime -> date only
    - How can I ensure that tpep\_pickup\_datetime returns date value only?
      * Using the date\_trunc()

|  |
| --- |
| select date\_trunc('day', tpep\_pickup\_datetime) as tpep\_pickup\_date  ,count(\*)  from qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz" yttdg  group by date\_trunc('day', tpep\_pickup\_datetime) |

Output: Two columns: Date truncated to day of the year, Count associated with the date

Question: What is the average daily trip count in 2018?

* Grain:
  + Year (top level)
  + Aggregate daily trip count

Note: Try to name CTE as explicitly as possible.

|  |
| --- |
| EXTRACT (YEAR FROM tpep\_pickup\_datetime) |

Input:

* Target extract date grain (ex: year, quarter, day, etc.).
* Timestamp column
  + If not timestamp use TIMESTAMP prefix to cast it as the timestamp

|  |
| --- |
| **SELECT** EXTRACT(**YEAR** **FROM** **TIMESTAMP** '2016-12-31 13:30:15'); |

Question: The average daily trip count in 2018?

* grain / row: year
  + aggregate daily trip count

|  |
| --- |
| with daily\_trip\_count as (  select  date\_trunc('day', tpep\_pickup\_datetime) as tpep\_pickup\_date ,  count(\*) trip\_count  from  qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz" yttdg  where  extract(year from tpep\_pickup\_datetime)= 2018  group by  date\_trunc('day', tpep\_pickup\_datetime) )  select  avg(trip\_count)  from  daily\_trip\_count |

Inner Query: Gives the aggregate daily trip count.

Outer Query: Calls the average function and averages those values using the information from the inner query. Sum of daily observations/ Number of days (366)

Question: The average daily distinct trip count in 2018?

* Grain/row: year
  + Aggregate distinct daily trip count
    - Distinct trips

|  |
| --- |
| with distinct\_trips as (  select  distinct \*  from  qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz"  where  extract(year from tpep\_pickup\_datetime)= 2018 ),  daily\_trip\_count as (  select  date\_trunc('day', tpep\_pickup\_datetime) as tpep\_pickup\_date ,  count(\*) trip\_count  from  distinct\_trips  where  extract(year from tpep\_pickup\_datetime)= 2018  group by  date\_trunc('day', tpep\_pickup\_datetime) )  select  avg(trip\_count)  from  daily\_trip\_count |

Step 1

* Using the distinct command, we established unique rows.
* Using the WHERE clause, we filtered out records that pertain to 2018 only.

Step 2

* We truncated the date values to obtain a daily column
* Used the count(\*) to calculate distinct trips.

Step 3

* Using the average function, we gained the average daily distinct count.

We essentially broke it into 3 steps and developed a CTE on that basis.

* Establish distinct\_trips
  + Referenced distinct\_trips in a subsequent CTE daily\_trip\_count
* We exited the CTE and referenced daily\_trip\_count with the average function.

Question: Average daily distinct trips as it pertains to April 2018

* grain / row: year
  + aggregate distinct daily trip count
    - distinct trips

|  |
| --- |
| select  distinct \*  from  qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz"  where  extract(year from tpep\_pickup\_datetime)= 2018  and extract(month from tpep\_pickup\_datetime)= 4 |

Note:

*Order:* [*https://www.postgresql.org/docs/9.5/sql-select.html*](https://www.postgresql.org/docs/9.5/sql-select.html)*: see the description for the order*

Step 1

* When a SQL statement is being executed, the order at which it is being executed is FROM first.
  + Identifying target object you are referencing
* The WHERE clause restricting the data followed by the SELECT and GROUP BY.
  + In the WHERE clause, we limited the number of records that the statement needs to do distinct on. This is why the query took a short amount of time.

Lecture 07

03/18/2021

**Order of operations in a query**: See the [Postgres documentation](https://www.postgresql.org/docs/9.5/sql-select.html).

1. Evaluate CTE
2. From
3. Where
   1. The smaller the query, the faster it will run.
   2. If you are running into performance issues, make a tighter where clause.
   3. Even if using the where clause causes a slightly slower query, all future queries/operations that use the filtered result will be much faster.
4. Group By
   1. The smaller the data you’re grouping, the faster the query will run.
5. Having
6. Select
7. Select Distinct
   1. Note when Aliasing of columns occurs in regards to the order of operations.
   2. Aliasing of columns occurs in the select clause; if we try to use the alias in the: from, where, or group by clause (or anything earlier), it won’t know what the alias references (SQL Error [42703]).
8. Union/Intersect/Except
   1. Each of these operations returns a set.
9. Order by
   1. Can use column aliases in this statement.
   2. Ordering is expensive
      1. If the data is filtered, there are less rows, and the ordering will be cheaper.
   3. [Ordering in Postgresql](https://wiki.postgresql.org/images/5/59/Sorting_through_the_ages.pdf) and how it is done.
   4. Maybe use [ASort](https://people.inf.ethz.ch/smilos/asort3.pdf) algorithm to get an approximate sort?
10. Limit
    1. Restricts the number of rows that are returned
11. Update

|  |
| --- |
| select \*,  tpep\_dropoff\_datetime :: timestamp as tpep\_dropoff\_timestamp  from "2018\_Yellow\_Taxi\_Trip\_Data\_gz"  where tpep\_dropoff\_datetime :: timestamp = timestamp'2018-04-12 20:52:08'  order by tpep\_dropoff\_timestamp |

|  |
| --- |
| select \*,  tpep\_dropoff\_datetime :: timestamp as tpep\_dropoff\_timestamp  from "2018\_Yellow\_Taxi\_Trip\_Data\_gz"  where tpep\_dropoff\_datetime :: timestamp >= timestamp'2018-04-12 00:00:00'  and tpep\_dropoff\_datetime :: timestamp < timestamp'2018-04-13 00:00:00'  order by tpep\_dropoff\_timestamp  limit 200 |

Observations and explaining the order of execution:

1. FROM
   1. "2018\_Yellow\_Taxi\_Trip\_Data\_gz" is knowing the \*, tpep\_dropoff\_datetime -> only the columns table
2. Where
   1. Filter out the rows that don’t match our criteria.
   2. The comparison will be executed.
   3. Converts the drop\_off time (varchar) and the boundary time (varchar) to a timestamp.
   4. Don’t compare timestamps as strings; there can be issues with the formatting[[1]](#footnote-0).
3. Select
   1. Select all columns using the \* operator, and we do a cast from varchar to timestamp, and then we alias the column name.
4. Order by
   1. Order the restricted results set by the aliased column name
5. Limit
   1. We only return the first 200 rows.
   2. This will be faster and can help with debugging[[2]](#footnote-1).

**Aggregation with View**

Once we have the logic down, we want to finalize that logic with a view

|  |
| --- |
| create view average\_daily\_distinct\_trip\_count as  with distinct\_trips as (  select  distinct \*  from  qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz"  where  extract(year from tpep\_pickup\_datetime)= 2018 ),  daily\_trip\_count as (  select  date\_trunc('day', tpep\_pickup\_datetime) as tpep\_pickup\_date ,  count(\*) trip\_count  from  distinct\_trips  where  extract(year from tpep\_pickup\_datetime)= 2018  group by  date\_trunc('day', tpep\_pickup\_datetime) )  select  avg(trip\_count)  from  daily\_trip\_count  ; |

When you create a view, what happens is that you associate the **logic** with a view, independent of the schema. I.e you have to recalculate the logic (sql script) each time on the associated table.

If we have a complex logic -> it will take longer for the view to execute

A view is not a materialized CTE; material is used in regard to whether the object is physical or not. Something that occupies a significant amount/calculable amount of storage.

In general, we want our views to be simple.

For recording purposes, it is not a good idea to use views.

Aggregation with Tables:

If we don’t like views, why not materialize the view into a table?

??

When to use a view:

* Time permits
  + Either quick, overnight it etc.
* When I only care about the results of the view once.
* Store **transformation logic**.
  + Cast a varchar to a timestamp, that is a transformation logic.
  + Calculations can be calculation logic.

|  |
| --- |
| create table average\_daily\_distinct\_trip\_count\_table as  with distinct\_trips as (  select  distinct \*  from  qcmath290.public."2018\_Yellow\_Taxi\_Trip\_Data\_gz"  where  extract(year from tpep\_pickup\_datetime)= 2018 ),  daily\_trip\_count as (  select  date\_trunc('day', tpep\_pickup\_datetime) as tpep\_pickup\_date ,  count(\*) trip\_count  from  distinct\_trips  where  extract(year from tpep\_pickup\_datetime)= 2018  group by  date\_trunc('day', tpep\_pickup\_datetime) )  select  avg(trip\_count)  from  daily\_trip\_count  ; |

To materialize a view, it will take a *very* long time the first time, but then each subsequent time it will be very quick since the data is there already.

Aggregation with Tables & Views

1. Build CTE
2. Build a View for the CTE
   1. CTE is just a string of code
   2. A View is a script/can be used in a select statement

CTE and View is based on the current status of the table.

* If the underlying table data is volatile, you should probably use a CTE/View
* If the data is slow moving, use a table
* If the data is big and fast, need to use an ETL

**Materializing Views**

create table <new table name> as

select \* from <view name>;

In a reporting setting, it is usually better to drop the table and then recreate the table from the view instead of updating. In an application ***DON’T DO THIS[[3]](#footnote-2)***. This pattern works in a limited fashion.

|  |
| --- |
| drop table average\_daily\_distinct\_trip\_count\_table\_from\_view;  create table average\_daily\_distinct\_trip\_count\_table\_from\_view as  select \* from average\_daily\_distinct\_trip\_count\_table; |

**Relationships in RDBMS**

* One-to-one(1:1)
  + A tuple in relation A can be associated with at most one (zero or 1) tuples in relation B
  + A tuple in relation B can be associated with at most one (zero or 1) tuples in relation A.
    - Student ID to ssn#
    - Student table, ssn table
      * Why not make it a new column in the student table?
        + Security reasons.
* One-to-many (1:n)
  + A tuple in relation A can be associated with any number (zero or more) tuples in relation B.
  + A tuple in relation B can be associated with at most one (zero or 1) tuples in relation A.
    - User to Grocery List
      * A user (relation A) can have many grocery lists
      * The grocery list (relation B) can only have 1 owner/user or be in the trash can (no owner).
* Many-to-one (n:1)
  + Same as one-to-many, but inverse relation A and B
* Many to Many (m:n)
  + A tuple in relation A can be associated with any number (zero or more) tuples in relation B
  + A tuple in relation B can be associated with any number (zero or more) tuples in relation A
    - Student ID to Interests
      * A student can have multiple interests, and an interest can be shared amongst multiple students

Lecture 08

03/25/2021

for transformation logic

|  |  |
| --- | --- |
| Use Table | Use View |
| * Underlying data is static * Underlying data is extremely big * When storage is **not** an issue | * Underlying data is very volatile (dynamic) * Limited storage |

\* Views are very CPU intensive

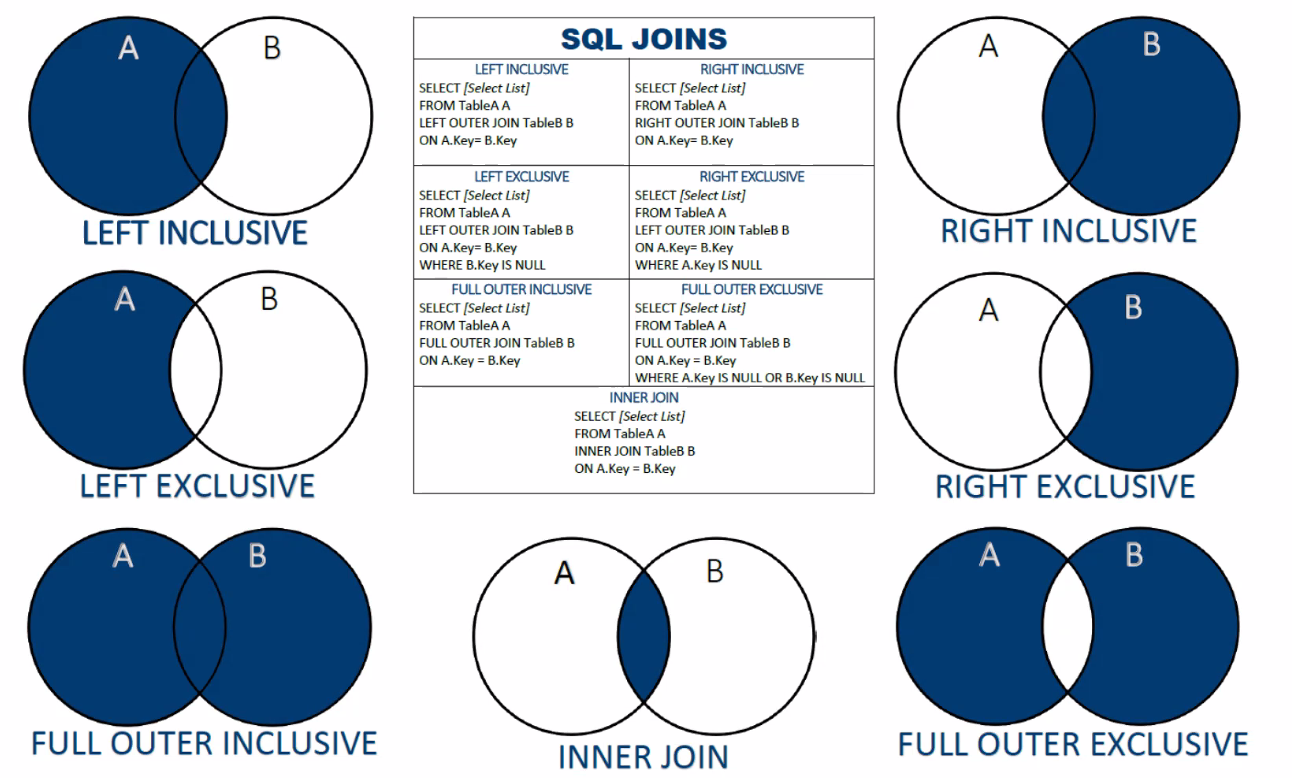
Ideal ETL pattern:

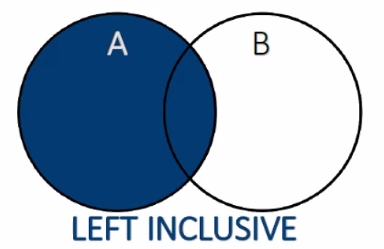
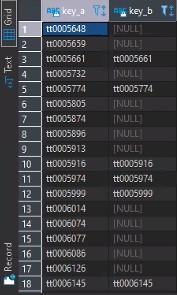
1. Stage the data in raw tables (as wide data types as possible)
   1. You don't want your process to get hung up
2. Use views for transformation (coercion of data type)
3. Use these views to create tables
   1. Data types are defined correctly
4. Export data into .csv
   1. Why? → .csv files are easily consumed by PowerBI

\* ETL : Extract, Transform, Load

Schemaless Data vs. Schema Data  
  
**Schemaless Data:** No table or data types. Everything is varchar. Has high query ability. Preferred by web developers due to not needing version control.   
  
**Schema Data**: Schema defined first. high level view of the structure and relationships.

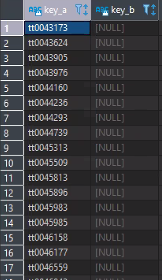
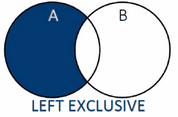
SQL JOINS





|  |
| --- |
| select a.tconst key\_a , b.tconst key\_b  from imdb.public.title\_basic a  left outer join imdb.public.title\_ratings b  on a.tconst = b.tconst; |

* Key A has all the values of tconst from title\_basic
* Key B has only values if title\_basic.tconst = title\_ratings.tconst



|  |
| --- |
| select a.tconst key\_a , b.tconst key\_b  from imdb.public.title\_basic a  left outer join imdb.public.title\_ratings b  on a.tconst = b.tconst  where b.tconst is null  ; |

* All the values in A that are not in B; values that are *exclusive* to A

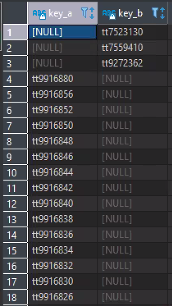
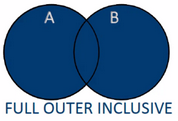
This is equivalent to the following logic:

|  |
| --- |
| select a.tconst key\_a,  from imdb.public.title\_basic a  where a.tconst not in (select tconst key\_b from imdb.public.title\_ratings b); |

But from a performance standpoint the *join* is better than the *in* clause.

|  |
| --- |
| select a.tconst key\_a,  from imdb.public.title\_basic a  where a.tconst != 'tt0000014' or a.tconst != 'tt0000020'...; |

The cardinality of b.tconst is going to be the number of *or* statements that needs to be executed.



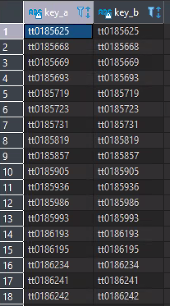
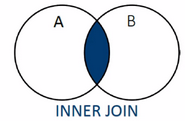
|  |
| --- |
| select a.tconst key\_a , b.tconst key\_b  from imdb.public.title\_basic a  full outer join imdb.public.title\_ratings b  on a.tconst = b.tconst  order by key\_a desc  ; |

- Obtaining all values from both tables, whether they are exclusive to either table or shared within both.

- Obtaining all values from table B even if it is not in table A.

As you can see, there are three values in table B that are not in table A so a primary key cannot be established.

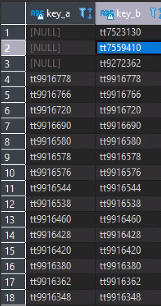
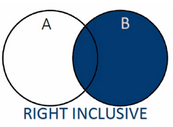
- For a good foreign key/primary key candidate, every value of the child (*Table B*) should exist in the parent (*Table A*).



|  |
| --- |
| select a.tconst key\_a , b.tconst key\_b  from imdb.public.title\_basic a  inner join imdb.public.title\_ratings b  on a.tconst = b.tconst; |

|  |
| --- |
| SELECT \*  FROM imdb.public.title\_ratings tr  WHERE tconst LIKE 'tt7523130'; |

- Selects all values from both tables as long as there is a match between the columns.



|  |
| --- |
| select a.tconst key\_a , b.tconst key\_b  from imdb.public.title\_basic a  right outer join imdb.public.title\_ratings b  on a.tconst = b.tconst  order by key\_a desc; |

- Reverse of *Left Inclusive.*

- Obtains all values from table B and all values that are in A as well.

Lecture 09

04/08/2021

Cross Join:

|  |
| --- |
| select tb.tconst, tr.tconst from  imdb.public.title\_basic tb  cross join imdb.public.title\_ratings tr |

- The two keys tconst in title\_basics and tille\_ratings are going to be compared

- Each tuple in title\_basics is going to be repeated or paired with each and every tuple in title\_ratings. You are going to be blowing up the space.

- When you want to create a time series product, a cross product will be useful.

- There are 7,619,417 tuples in title\_basics and 1,123,371 tuples in title\_ratings. The cross product of the two tables will give … tuples.

Self Join:

|  |
| --- |
| select \* from  imdb.public.title\_basic tb\_a  left join imdb.public.title\_basic tb\_b on tb\_a.tconst = tb\_b.tconst; |

Both queries produce the same output

|  |
| --- |
| select tb\_a.\*,tb\_b.\* from  imdb.public.title\_basic tb\_a  left join imdb.public.title\_basic tb\_b on tb\_a.tconst = tb\_b.tconst; |

- You join the table to itself that is why it is a self join.

- To do a self join, you take a table and do some kind of join (could be left, inner, right, etc.) and join it to itself on the same key.

- The number of columns this table will have is double of the number of columns in title\_basics since we have select \* in the example.

|  |
| --- |
| select tb\_a.tconst as tconst\_a, tb\_a.originaltitle as originaltitle\_a, tb\_a.runtimeminutes as runtimeminutes\_a,  tb\_b.tconst as tconst\_b, tb\_b.runtimeminutes as runtimeminutes\_b, tb\_b.originaltitle as originaltitle\_b from  imdb.public.title\_basic tb\_a  left join imdb.public.title\_basic tb\_b on cast(tb\_a.runtimeminutes as integer)  < cast(tb\_b.runtimeminutes as integer)  where tb\_a.tconst ='tt0009143'  ; |

- This query uses the nested join algorithm because we are using the less than (<) operation.

- We are joining on tb\_a.runtimeminutes, where it needs to be less than tb\_b.runtimeminutes. This means that we are finding a specific movie’s runtime (because of the where clause), where it is less compared to other movies’ runtime.

**Problem**: For any combination of two runtime of movies give all the movies where the runtime is less than or equal to 100.

Solution 1:

|  |
| --- |
| with cte\_cartesian as (  select  tb\_a.tconst as tconst\_a,  tb\_a.originaltitle as originaltitle\_a,  tb\_a.runtimeminutes as runtimeminutes\_a,  tb\_b.tconst as tconst\_b,  tb\_b.runtimeminutes as runtimeminutes\_b,  tb\_b.originaltitle as originaltitle\_b  from  imdb.public.title\_basic tb\_a, --cross join  imdb.public.title\_basic tb\_b  ), cte\_cartesian\_sum as (  select \*, cast(runtimeminutes\_b as integer) + cast(runtimeminutes\_a as integer) as combined\_runtime from cte\_cartesian  ), cte\_solution as (  select \* from cte\_cartesian\_sum  where combined\_runtime<=100  ) select distinct \* from cte\_solution; |

- Steps:

1. You need to create a combination of all movie runtime. This will be a cartesian self join. (equivalent to a cross join)
2. You need to create another join, where you have to join the tile\_basics table to the cartesian table and carry out the comparison.

- There will be repetition in the result if you do not include distinct to the final table result (cte\_solution).

Solution 2:

|  |
| --- |
| select distinct  tb\_a.tconst as tconst\_a,  tb\_a.originaltitle as originaltitle\_a,  tb\_a.runtimeminutes as runtimeminutes\_a,  tb\_b.tconst as tconst\_b,  tb\_b.runtimeminutes as runtimeminutes\_b,  tb\_b.originaltitle as originaltitle\_b,  cast(tb\_b.runtimeminutes as integer) + cast(tb\_a.runtimeminutes as integer) as combined\_results  from  imdb.public.title\_basic tb\_a  left join imdb.public.title\_basic tb\_b on cast(tb\_b.runtimeminutes as integer) + cast(tb\_a.runtimeminutes as integer) <=100  where tb\_b.runtimeminutes is not null or tb\_a.runtimeminutes is not null; |

- This query does not omit null values, so you need to add a where clause to eliminate those case.

Analysis:

* The two solutions differ in comparison. So, in solution 2 the comparison happens in the join clause, where in one step you add the two values together and carry out the comparison and it only brings a value if the join comparison is satisfied. In the first solution, you create the combination independent, and then compare if the two movies will give you a sum of runtime less than or equal to 100 and you filter at the end.
* Both solutions take a long time to execute.

Recursive Join:

* You can reference inside the CTE by a recursive join

**Problem**: Who is the CEO? (aka who doesn't report to anybody?)

Created Table for this problem

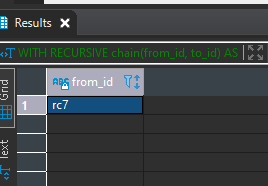
|  |
| --- |
| CREATE TABLE Table1  ("ID1" text, "ID2" text)  ;    INSERT INTO Table1  ("ID1", "ID2")  VALUES  ('vc1', 'vc2'),  ('vc2', 'vc3'),  ('vc3', 'vc4'),  ('vc4', 'rc7')  ; |

Solution:

|  |
| --- |
| WITH RECURSIVE chain(from\_id, to\_id) AS (  SELECT NULL, 'vc2'  UNION  SELECT c.to\_id, t."ID2"  FROM chain c  LEFT OUTER JOIN Table1 t ON (t."ID1" = to\_id)  WHERE c.to\_id IS NOT NULL  )  SELECT from\_id FROM chain WHERE to\_id IS NULL; |

- ‘vc2’ can be any value, this is the anchor in this problem to help you start off with the recursion.

Output:



- This value has no body in the hierarchy to report to

One-to-one Relationship:

|  |
| --- |
| select count(distinct tb.tconst) as dst\_cnt\_tb\_tconst,  count(tb.tconst) as cnt\_tb\_tconst,  count (distinct tr.tconst) as dst\_cnt\_tr\_tconst,  count (tr.tconst) as cnt\_tr\_tconst  from imdb.public.title\_basic tb  inner join imdb.public.title\_ratings tr on tb.tconst = tr.tconst |

- When we only have one value for each table for the attribute key that is considered one-to-one relationship.

- To see if two tables have a one-to-one relationship, you do an inner join on the two tables. If the distinct count and the count in tb.tconst has the same value and the distinct count and the count in tr.tconst, then it is a one-to-one relationship.

- You could also do a full outer join, and if it is the same as the inner join output that is considered good as well, and shows a one-to-one relationship.

Lecture 10

04/15/2021

* AND takes both sides of the operator and both sides must be the same to be true.
  + If both values are null, then it will return null.
  + If one of the values is either true/false and the other null, then it will return null.

\*AND is bias towards false

* OR evaluates both sides and returns the UNION.
  + If both values are true, then it returns true. The same works for false.
  + If one of the values is true and the other false/null, then it returns true.
  + If both sides of the operator are null, then it returns null.

\*OR is bias towards true

* NOT returns the opposite of a condition. Think of multiplying by -1.

\*mirror image

|  |
| --- |
| create table employee (emp\_id integer, emp\_name varchar, on\_call boolean);  create table facility (emp\_id integer, facility\_name varchar );  insert into employee  select 1, 'John', true  union  select 2, 'Bob', false  union  select 3, 'Ed', false  union  select 4, 'Stu', true  insert into facility  select 1, 'warehouse'  union  select 2, 'office'  union  select 3, 'office'  union  select 4, 'store' |

- Create two tables: employee and facility

- Insert values in both tables

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call,  from  employee e,  facility f  where  e.emp\_id = f.emp\_id -- predicate a  and  f.facility\_name in('warehouse','office') -- predicate b    --a and b |

- Predicate a is being used as an equal operator. We are trying to make sure that they have the same values.

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call,  from  employee e  left join facility f on e.emp\_id = f.emp\_id |

- Left join would return every value in the employee table and facility values that are true

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call,  from  employee e, facility f  where  e.emp\_id = f.emp\_id -- predicate a  and  f.facility\_name ='warehouse' -- predicate b  or f.facility\_name ='office' -- predicate c |

- Predicate a is using an equal operator for e.em\_id and f.emp\_id

- Predicate b is using an equal operator to check the facility name values is equal to ‘warehouse’

- Predicate c is using an equal operator to check the facility name values is equal to ‘office’

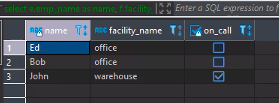
|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call  from  employee e, facility f  where  e.emp\_id = f.emp\_id -- predicate a  and  f.facility\_name ='warehouse' -- predicate b |



- If the id in the employee table is the same as the id in the facility table, then the name of the employee in the warehouse will be in the output.

-John’s id matches in both tables and predicate b is true

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call  from  employee e inner join facility f  on  e.emp\_id = f.emp\_id  where  on\_call = true -- predicate a  and f.facility\_name ='warehouse' -- predicate b  or f.facility\_name ='office' -- predicate c |

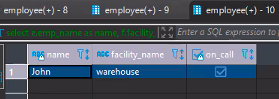


- Where clause with AND and OR with no parenthesis

- It will output all the values that complies with the facility names (warehouse, office) but it outputs all the on\_call values as well.

- We want the on\_call values that are true

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call  from  employee e inner join facility f  on  e.emp\_id = f.emp\_id  where  on\_call = true -- predicate a  and (f.facility\_name ='warehouse' -- predicate b  or f.facility\_name ='office') -- predicate c |

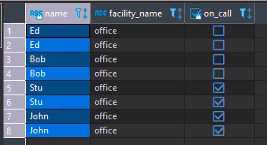


A AND (B OR C)

- We want the facility name for warehouse or facility name for office.

-With parenthesis, it will output one the value that is true (that complies with predicate a)

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call  from  employee e inner join facility f  on  e.emp\_id = f.emp\_id  where  on\_call = true -- predicate a  and f.facility\_name ='warehouse' -- predicate b  or f.facility\_name ='office' -- predicate c |



- Inner join would return matching values in both tables.

- Since there is no parenthesis in the where clause, it outputted all the on\_call

|  |
| --- |
| select  e.emp\_name as name,  f.facility\_name as facility\_name,  e.on\_call as on\_call  from  employee e inner join facility f  on  e.emp\_id = f.emp\_id  where  on\_call = true -- predicate a  and (f.facility\_name ='warehouse' -- predicate b  or f.facility\_name ='office') -- predicate c |

- In this block of code, because there is the presence of parenthesis, then it will return on\_call values that are true.

Lecture 10

04/22/2021

HW08 coding interview queries:

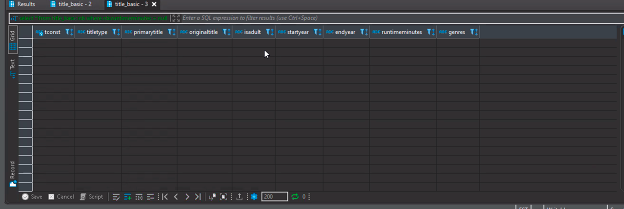
|  |
| --- |
| -- Problem 1 (algebra)  select primaryname as "Name", SUM(runtimeminutes) as "Total Runtime" from  xf\_title\_basic cross join xf\_title\_principals cross join xf\_name\_basics where  xf\_title\_basic.tconst = xf\_title\_principals.tconst and xf\_title\_principals.nconst = xf\_name\_basics.nconst and  primaryname like '%Nicolas%Cage%' and titletype = 'movie'  group by primaryname;  --1 row(s) fetched 41.8s |

|  |
| --- |
| -- Optimised using inner join  select primaryname as "Name", SUM(runtimeminutes) as "Total Runtime" from  xf\_title\_basic inner join xf\_title\_principals on  xf\_title\_basic.tconst = xf\_title\_principals.tconst inner join  xf\_name\_basics on xf\_title\_principals.nconst = xf\_name\_basics.nconst where  primaryname like '%Nicolas%Cage%' and titletype = 'movie'  group by primaryname;  --1 row(s) fetched 28.7s |

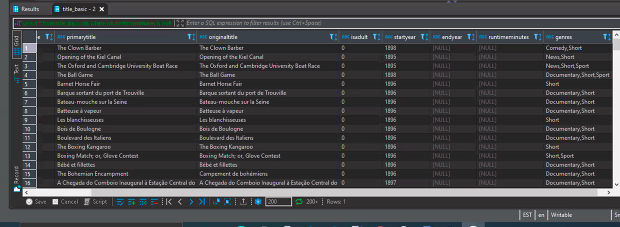
Difference between “IS NULL” and “ = Null”:

|  |
| --- |
| select \*  from title\_basic nb  where  nb.runtimeminutes=null; |

-- no result because “ = null” is always false.



|  |
| --- |
| select \*  from title\_basic nb  where  nb.runtimeminutes is null; |



|  |
| --- |
| select null = null; |

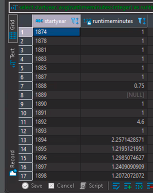
Is it a good idea to have NULL as your license plate?

|  |  |  |
| --- | --- | --- |
| License Plate | Date Time | Has E-ZPass |
| ‘ABC 123’ | 2021-04-12 | True |
| NULL |  |  |
| ‘NULL’ |  |  |

Analytical functions:

* Used to

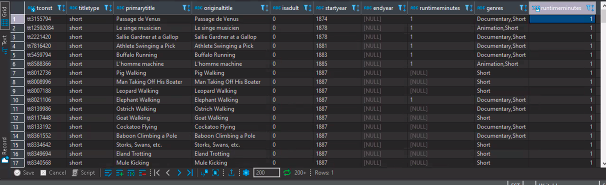
|  |
| --- |
| select  startyear,  avg(runtimeminutes::integer) as runtimeminutes  from  title\_basic nb  group by  startyear; |



Doing same query as above using window functions:

|  |
| --- |
| select  \*,  avg(runtimeminutes::integer) over (partition by startyear) as runtimeminutes  from  title\_basic nb; |

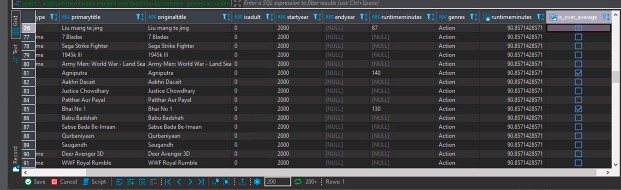
* Allows you to calculate the average for every tuple and populate back in table.
* No need to use “group by” clause



We can modify above query as:

Which movies have runtime that are above the average run time of the year and genres after startyear 2000?

|  |
| --- |
| select  \*,  avg(runtimeminutes::integer) over (partition by startyear,  genres) as runtimeminutes,  case  when avg(runtimeminutes::integer) over (partition by startyear, genres)<runtimeminutes::integer then true  else false  end as is\_over\_average  from  title\_basic nb  where  startyear >= '2000' ; |



Q. Which movies have runtime that are above average runtime per year and genres?

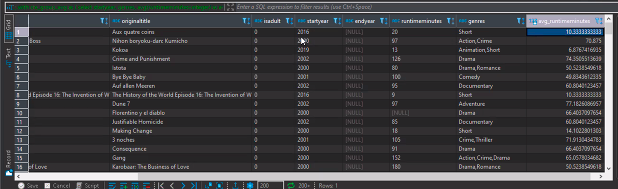
Step 1: without CTE, have a query grouping by startyear and genre.

Filter the data first to make it smaller.

|  |
| --- |
| select  startyear, genres, avg(runtimeminutes::integer) as avg\_runtimeminutes  from title\_basic nb  where startyear::integer>=2000  group by  startyear, genres; |

Step 2: Do inner join above query with cte on startyear and genres

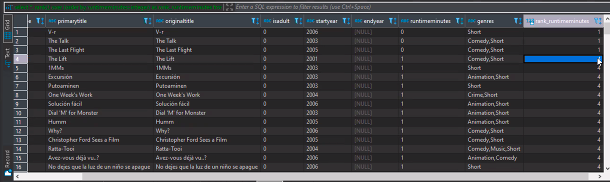
|  |
| --- |
| with cte\_group\_avg as (  select  startyear,  genres,  avg(runtimeminutes::integer) as avg\_runtimeminutes  from  title\_basic nb  where  startyear::integer >= 2000  group by  startyear,  genres )  select  nb.\*,  cga.avg\_runtimeminutes  from  title\_basic nb  inner join cte\_group\_avg cga on  nb.startyear = cga.startyear  and nb.genres = cga.genres; |



Window Function rank():

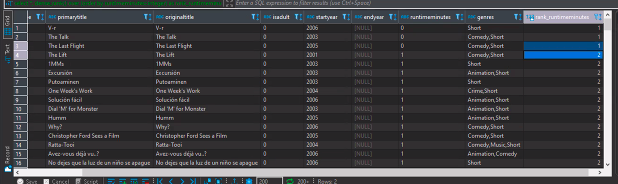
-- Give numerical rank to every run time based on its length

|  |
| --- |
| select  \*,  rank() over (order by runtimeminutes::integer) as rank\_runtimeminutes  from  title\_basic nb  where  startyear::integer >= 2000; |



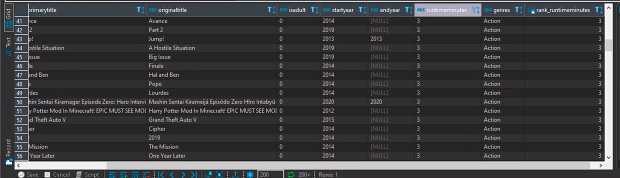
* The rows that have the same values will receive the same **rank**.
* Next rank will not be next number. There will be a gap.
* Use dense\_rank() to have no gaps within ranks.

|  |
| --- |
| select  \*,  dense\_rank() over (order by runtimeminutes::integer) as rank\_runtimeminutes  from  title\_basic nb  where  startyear::integer >= 2000; |



Use partition by genres to start the counter of rank within the genre.

|  |
| --- |
| select  \*,  dense\_rank() over (partition by genres order by runtimeminutes::integer) as rank\_runtimeminutes  from  title\_basic nb  where  startyear::integer >= 2000; |

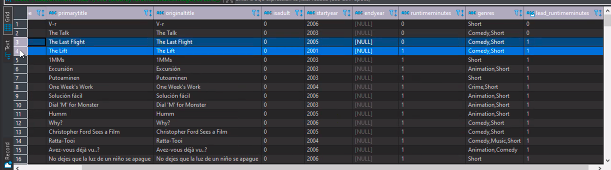


Lead() operator:

Inputs: A column and a offset

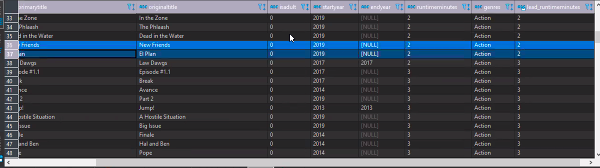
Offset is the number of rows you want to forward.

|  |
| --- |
| select  \*,  lead(runtimeminutes, 1) over (order by runtimeminutes::integer) as LEAD\_runtimeminutes  from  title\_basic nb  where  startyear::integer >= 2000; |



Use partition by to reset the counter:

|  |
| --- |
| select  \*,  lead(runtimeminutes, 1) over (partition by genres order by runtimeminutes::integer) as LEAD\_runtimeminutes  from  title\_basic nb  where  startyear::integer >= 2000; |



AWS:

* RDS: Relational Data Service. Allows you create a new database. Pick any (PostgreSql preferred). Fill out Database id, password, pick db instance(small and free).
* VPC: Virtual Private Cloud. Equivalent to local network. To access your database from a local machine make sure your local machine has access to VPC Before creating a db make sure to create a VPC. Refer to class lecture recording for how to create a VPC.
* EC2: Elastic Compute Cloud. Equivalent to machine on the cloud (Computing machine).
* S3:

1. Also, if there is an error with the format of one of the strings, the error won’t be thrown. [↑](#footnote-ref-0)
2. I found this very useful. [↑](#footnote-ref-1)
3. Don’t you dare quote Prof. Zombory on this as good practice in your application. [↑](#footnote-ref-2)