Chapter 10

Analytic Functions & CTE's

Contents

1 Analytic Functions

- Analytic (sometimes called “window” or “partition” functions) functions were designed to simplify many common complex joins.
- There are a few different use cases that they can greatly simplify.
- As an example, lets consider the case of computing the *percentage* of traffic which pays by cash by hour and plaza in the inbound direction on November 10th, 2016. To do this we would need to take our original data and then join it against the correct sum. In other words we want to return 24 columns for each plaza and the sum (vertically, across hour) should be equal to 1.
- In order to do this calculation we need to join our original data back onto the proper sum, as can be seen in the query below.

```
select
    plaza
    , hr
    , vehiclescash::float / ALLVech as pctperhr
    , vehiclescash
    , ALLVech
from
    (select plaza, hr, vehiclescash
        from cls.mta
        where mtadt = '2016-11-10'
        and direction = 'I') as lhs
left join
    (select plaza, sum(vehiclescash) as ALLVech
        from cls.mta
        where mtadt = '2016-11-10'
        and direction = 'I'
        group by 1) as rhs
using(plaza)
order by 1,2;


  plaza    hr    pctperhr    vehiclescash    allvech
-------  ----  ----------  --------------  ---------
      1     0  0.0230918              167       7232
      1     1  0.017146               124       7232
      1     2  0.00954093              69       7232
      1     3  0.00940265              68       7232
      1     4  0.0199115              144       7232
[...]
```

  but this construct is a bit cumbersome.

- For another example, consider wanting to create a rolling sum over each plaza day for the number of cars which use cash in the inbound direction.

```
select
    lhs.plaza, lhs.mtadt, lhs.hr, sum( rhs.vehiclescash ) as cum_sum
from
    (select plaza, hr, mtadt
            from cls.mta where direction = 'I') as lhs
left join
    (select plaza, hr, mtadt, vehiclescash
            from cls.mta where direction = 'I') as rhs
on
    lhs.plaza = rhs.plaza
    and lhs.hr >= rhs.hr
    and lhs.mtadt = rhs.mtadt
group by lhs.plaza, lhs.mtadt, lhs.hr
order by 1,2,3


  plaza  mtadt         hr    cum_sum
-------  ----------  ----  ---------
      1  2010-01-01     0        474
      1  2010-01-01     1       1191
      1  2010-01-01     2       1855
      1  2010-01-01     3       2450
      1  2010-01-01     4       2997
[...]
```

- These two examples have a set of common properties: we need to aggregate over our table while returning the original table. Doing this using the techniques we've seen in the past is cumbersome, so we can use Analytic (sometimes called window or partition functions) to solve them.

- Analytic functions use the following syntax:

```
function () over(
        partition by ______________
        order by ______________
        <WINDOW FRAME CLAUSE>
)
```

function can be one of any of our standard aggregate functions (SUM, COUNT, MAX, MIN, AVG) as well as a number of functions that can only be used as analytic functions.

There are a few pieces of the syntax:

1. The OVER() clause: This tells the database to expect a window function, rather than a standard aggregate function. This is required when using analytic functions.

2. The PARTITION BY clause: This clause tells the database how to break up the data. In other words, it is similar to a GROUP BY in that it tells the database that rows with the same values should be treated as a single entity or partition. The PARTITION BY clause is optional.

3. The ORDER BY clause: This clause works just as an ORDER BY in a normal SQL query works. It tells the database how to sort the data within each partition. The ORDER BY clause is optional. If an ORDER BY clause is present then the function is calculated in a running

fashion – e.g. as a running some from the start of the partition to the current row.

4. The WINDOW FRAME clause defines the region over which the function is calculated. It takes on a number of different forms though the most common is the rows between syntax:

```
ROWS BETWEEN ______________ AND ____________
```

the blanks would take on some of the following values:

   - UNBOUNDED PRECEDING: from the start of the partition
   - UNBOUNDED FOLLOWING: to the end of the partition
   - XX PRECEDING: XX rows preceding (inclusive)
   - XX FOLLOWING: XX rows following (inclusive)
   - CURRENT ROW: the current row

   In other words we can use this syntax to easily compute things like hourly moving averages or just smoothing.

- Lets use analytics functions to solve the two problems at the start of this section. To solve the first one we can do the following:

```
select
    plaza
    , hr
    , vehiclescash::float/sum(vehiclescash) over(partition by plaza) as pctperhr
    , vehiclescash
    , sum( vehiclescash) over(partition by plaza) as ALLVech
from
    cls.mta
where
    mtadt = '2016-11-10' and direction = 'I'
order by 1,2;


  plaza    hr    pctperhr    vehiclescash    allvech
-------  ----  ----------  --------------  ---------
      1     0  0.0230918              167       7232
      1     1  0.017146               124       7232
      1     2  0.00954093              69       7232
      1     3  0.00940265              68       7232
      1     4  0.0199115              144       7232
[...]
```

the OVER clause, which modifies the SUM function, tells the database that it is going to be computing an aggregate function, but *without the aggregation.* In other words, it will return the same value for each row.

- To be clear on what this is doing, let's consider only a single plaza (#1) and look at the hourly data for that day, including the analytic function:

```
select
    plaza
    , hr
    , vehiclescash
    , sum(vehiclescash) over(partition by plaza) as totalcars
from
    cls.mta
where
    mtadt = '2016-11-10'
    and direction = 'I'
    and plaza = 1;


  plaza    hr    vehiclescash    totalcars
-------  ----  --------------  -----------
      1     0             167         7232
      1     1             124         7232
      1     2              69         7232
      1     3              68         7232
      1     4             144         7232
      1     5             215         7232
      1     6             281         7232
      1     7             336         7232
      1     8             329         7232
      1     9             304         7232
      1    10             344         7232
      1    11             286         7232
      1    12             308         7232
      1    13             375         7232
      1    14             361         7232
      1    15             471         7232
      1    16             450         7232
      1    17             451         7232
      1    18             420         7232
      1    19             446         7232
      1    20             366         7232
      1    21             322         7232
      1    22             296         7232
      1    23             299         7232
```

The sum of vehiclescash on this subset is 7,232 – the exact number returned by the analytic function in the total cars column.

- To solve the cumulative sum problem we can use an analytic function in the following manner:

```
select
    plaza, mtadt, hr,
    sum(vehiclescash) over(
        partition by plaza, mtadt
        order by hr
        rows between unbounded preceding and current row) as cum_sum
from
    cls.mta
where
    direction = 'I'


  plaza  mtadt         hr    cum_sum
-------  ----------  ----  ---------
      1  2010-01-01     0        474
      1  2010-01-01     1       1191
      1  2010-01-01     2       1855
      1  2010-01-01     3       2450
      1  2010-01-01     4       2997
[...]
```

- To get more insight into the specifics of how analytic functions work, consider the following table:

GRP	ORD	NM	C0	C1	C2	C3
1	1	5	65	33	5	11
1	2	6	65	33	11	21
1	3	10	65	33	21	28
1	4	12	65	33	33	22
2	1	12	65	32	12	22
2	2	10	65	32	22	28
2	3	6	65	32	28	20
2	4	4	65	32	32	10

In this table the raw data is GRP, ORD and NM. In order to create columns C1, C2 and C3 we use the following syntax:

```
SUM(NM) OVER( ) as C0
SUM(NM) OVER( PARTITION BY GRP ) as C1
SUM(NM) OVER( PARTITION BY GRP ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as C1
SUM(NM) OVER( PARTITION BY GRP ORDER BY ORD ASC) as C2
SUM(NM) OVER( PARTITION BY GRP ORDER BY ORD ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS C3
```

- **NOTE:** The default behavior of different analytic functions when using different sets of arguments can lead to issues. I have a few cases memorized, but I'd recommend being as inclusive as possible with the arguments. In the example above, `C1` is presented twice, both would return the same numbers, but the second makes it more clear as to what is happening.

- All of the aggregate functions we used in the past (MAX, MIN, COUNT and AVG) can be used with analytic functions. For example, to create a moving average based on the last 4 hours of data in the MTA dataset we could do the the following. Note that the ROWS BETWEEN function is inclusive, so it will average over four hours in the below.

```
select
    plaza
    , direction
    , hr
    , mtadt
    , vehiclescash + vehiclesez as totalcars
    , avg(vehiclescash + vehiclesez)
        OVER(PARTITION by plaza, direction
            ORDER BY mtadt asc, hr asc
            ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) as hravg
from
    cls.mta;


  plaza  direction      hr  mtadt         totalcars    hravg
-------  -----------  ----  ----------  -----------  -------
      1  I               0  2010-01-01          889    889
      1  I               1  2010-01-01         1419   1154
      1  I               2  2010-01-01         1223   1177
      1  I               3  2010-01-01         1075   1151.5
      1  I               4  2010-01-01          945   1165.5
[...]
```

- If we use a WINDOW FRAME clause without an ORDER BY then the row order returned is arbitrary.
- The only aggregate function not allowed is COUNT(DISTINCT) which cannot be used with the OVER() clause.
- There are a number of non-aggregate functions that can be used as analytic functions:
  1. LAG() and LEAD(): These functions return the value of a column from a preceding or following row.
  2. FIRST_VALUE(), LAST_VALUE() and NTH_VALUE() : These function return the first, last, or more generally, the nth value within a partition. Note that the NTH_VALUE function takes not only a column name, but a positional argument starting at 1. It will return null if there aren't enough values within the partition.
  3. NTILE(): This function handles percentiles.
  4. ROW_NUMBER(): This function returns the row number based on the criteria established in the clause. Note that the function ROW_NUMBER() fails without an OVER clause.
  5. RANK(): Returns the rank of a particular observation
  6. DENSE_RANK(): Returns the dense rank of a particular observation.
- The commands `ROW_NUMBER`, `RANK` and `DENSE_RANK` behave similarity when the data being sorted is unique. If the data is not unique, however, these commands behave differently, as demonstrated in the Table 10.1
- These functions allow us to easily answer a number of questions (without using JOIN). Such as what is the correlation between the absolute (nominal) change in vehicles paying cash and the absolute (nominal) change in vehicles paying by EZ-pass?

ID	ROW_NUMBER	RANK	DENSE_RANK
1	1	1	1
1	2	1	1
1	3	1	1
1	4	1	1
2	5	5	2
2	6	5	2
3	7	7	3

Table 10.1: Ranking function differences when ordering by the ID columns

```
SELECT
    CORR( CashDiff, EZDiff ) as DiffCor
FROM
    (SELECT
        LAG(vehiclesEZ) OVER(PARTITION BY plaza
                ORDER BY mtadt ASC, hr asc)
            - vehiclesEZ AS EZDiff
        , LAG(vehiclescash) OVER(PARTITION BY plaza
                ORDER BY mtadt ASC, hr asc)
            - vehiclescash AS CashDiff
    FROM
        cls.mta ) as innerQ;


  diffcor
---------
 0.744516
```

- These types of functions are evaluated *after* with SELECT after GROUP, JOIN, WHERE and HAVING. This means that you can't refer to them within those functions. If you want to filter on a window function it must be contained within a subquery.
- An important caveat when using these functions, as said from the documentation (emphasis mine):

  > 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.

  This is weird:

```
select
    plaza, mtadt, hr, direction
    , vehiclesez
    , sum(vehiclesez)
        over(partition by plaza order by mtadt, hr
                rows between unbounded preceding
                        and current row) as runningS1
    , sum(vehiclesez)
        over(partition by plaza order by mtadt, hr ) as runningS2
from
        cls.mta
where plaza = 1 and mtadt = '2010-01-01' and hr < 3;


  plaza  mtadt         hr  direction      vehiclesez    runnings1    runnings2
-------  ----------  ----  -----------  ------------  -----------  -----------
      1  2010-01-01     0  I                     415          415          801
      1  2010-01-01     0  O                     386          801          801
      1  2010-01-01     1  I                     702         1503         2037
      1  2010-01-01     1  O                     534         2037         2037
      1  2010-01-01     2  I                     559         2596         3106
[...]
```

This is weird because when you omit ROWS BETWEEN, the running sum is computed as if rows which have similar values in the partition are the same. The same query however, with a ROWS BETWEEN clause computes a running sum while ignoring the duplicate rows.

2 Using Analytic Functions with Transaction Data

- In this section we return to trying to understand the revenue behavior of our soap transaction data. Just like before we are going to use the notion of a cohort to help our analysis.
- Consider the data in Table 9.1 which contains information on users who were making certain transactions.
- Let's begin by calculating the revenue per user by locale and also by their install time period.
- We didn't even attempt this in the previous section because it involved so many joins! Using Analytic functions allows us to skip many of those issues!

```
select
    cohort
    , locale
    , count(distinct userid) as numusers
    , sum(case when trans_dt::date <= (cohort + '1 month'::interval)::date
        then amt else 0 end ) as mon_0_amt
    , sum(case when trans_dt::date <= (cohort + '2 month'::interval)::date
        then amt else 0 end ) as mon_1_amt
    , sum(case when trans_dt::date <= (cohort + '3 month'::interval)::date
        then amt else 0 end ) as mon_2_amt
from
( select
    first_value(locale) over(partition by userid order by trans_dt asc ) as locale
    , date_trunc('month', first_value(trans_dt)
        over(partition by userid order by trans_dt asc))::date as cohort
    , amt, userid, trans_dt
from
    cls.trans ) as innerQ
GROUP BY 1,2


cohort      locale      numusers    mon_0_amt    mon_1_amt    mon_2_amt
----------  --------  ----------  -----------  -----------  -----------
2016-01-01  Canada          4706       162262       167287       223375
2016-01-01  Mexico          3920       186854       190914       224451
2016-01-01  U.S.           12676       542198       599027       637072
2016-02-01  Canada          4334       147535       152976       206147
2016-02-01  Mexico          3602       171472       175171       208097
[...]
```

- Let's calculate the percentage of revenue that each transaction represents for each userid (how would we do this without Analytic Functions?):

```
select
    userid, trans_dt, amt
    , amt/sum( amt) over(partition by userid) as pct
from
    cls.trans


  userid  trans_dt      amt    pct
--------  ----------  -----  -----
       1  2016-05-09  23.98    1
       2  2018-08-25  12.99    1
       3  2017-03-05  43.16    0.5
       3  2017-04-05  43.16    0.5
       4  2016-02-28  59.95    1
[...]
```

- I want to calculate the percentage likelihood that a person who has made X purchases makes another one. There are a number of different ways that this can be done, but we can use analytic functions:

```
select
    transNum
    , sum( case when transNum = totalTrans then 1 else 0 end)::float
    / count(1) as pct
    , count(1) as numerator
from
(select
    row_number() over(partition by userid order by trans_dt) as transNum
    , count(1) over(partition by userid) as totalTrans
from
    cls.trans) as innerQ
group by 1
order by 1;


  transnum       pct    numerator
----------  --------  -----------
         1  0.529178       574289
         2  0.538423       270388
         3  0.58621        124805
         4  0.644172        51643
         5  0.686276        18376
[...]
```

3 Common Table Expressions ("CTE")

- A relatively new piece of SQL syntax is WITH, which allows for tables to be defined and used repeatedly within a query. These are called Common Table Expressions. CTE are incredibly powerful ways of writing queries, but they can come with significant downsides (as we will discuss later).

  From PostgreSQL's documentation:

  > A useful property of WITH queries is that they are evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, expensive calculations that are needed in multiple places can be placed within a WITH query to avoid redundant work. Another possible application is to prevent unwanted multiple evaluations of functions with side-effects. However, the other side of this coin is that the optimizer is less able to push restrictions from the parent query down into a WITH query than an ordinary sub-query. The WITH query will generally be evaluated as written, without suppression of rows that the parent query might discard afterwards. (But, as mentioned above, evaluation might stop early if the reference(s) to the query demand only a limited number of rows.)

- The motivation for CTE is that they can increase readability in query by defining a table at the start of your query which only exists for the duration of the query.
- The WITH clause is used to start a CTE and it basically sets up a derived table that can be used in the query.
- Consider the following example:

```
with only_inbound as (select * from cls.mta where direction = 'I')

select * from only_inbound limit 100;


  plaza  mtadt         hr  direction      vehiclesez    vehiclescash
-------  ----------  ----  -----------  ------------  --------------
      2  2013-10-14    16  I                    2469             336
      2  2013-10-14    17  I                    2853             425
      2  2013-10-14    18  I                    2575             394
      2  2013-10-14    19  I                    2422             344
      2  2013-10-14    20  I                    1989             339
[...]
```

The basic syntax of the query is that we define a table via a query at the start using a `WITH` clause. This table does not have a schema and can only be referenced within that query.

- We can use CTEs with multiple queries by separating them with commas:

```
with
    only_inbound as (select * from cls.mta where direction = 'I')
    , only_outbound as (select * from cls.mta where direction = 'O')

select
    plaza, mtadt, hr
    , only_inbound.vehiclesez as inbound_ez
    , only_outbound.vehiclesez as outbound_ez
 from
    only_inbound
join
    only_outbound
using( plaza, mtadt, hr)

limit 10;


  plaza  mtadt         hr    inbound_ez    outbound_ez
-------  ----------  ----  ------------  -------------
      2  2013-10-14    17          2853           2116
      2  2013-10-13    11          2960           2081
      2  2013-10-12    17          2847           2433
      2  2013-10-10     1           189            177
      2  2013-10-10     3           118            140
[...]
```

- Where I find CTEs to be useful is when there are multiple layers of logic that need to be implemented. By using a CTE I can break up that application logic into separate pieces that are easier to read.

4 CTEs with the transaction data

- To use the WITH clause you specify a table name and then use AS. It is done before the SELECT in the query. For example, the following creates a table that only looks at unit transactions from the United States. We then use this figure out the average order value of these transactions:

```
with USUnits as (
     select * from cls.trans
     where locale = 'U.S.' and type = 'Units')

select avg(amt) as AOV from USUnits;



     aov
-------
43.3045
```

- Consider the following, more useful, example which creates an LTV dataset which has the first value of the local of purchase.

```
with LTVData as (select
    first_value(locale) over(partition by userid order by trans_dt asc ) as locale
    , date_trunc('month', first_value(trans_dt)
        over(partition by userid order by trans_dt asc))::date as cohort
    , amt, userid, trans_dt
from
    cls.trans )

select * from LTVData where locale = 'U.S.' limit 100;


locale    cohort        amt    userid  trans_dt
--------  ----------  -----  --------  ----------
U.S.      2016-05-01  23.98         1  2016-05-09
U.S.      2017-03-01  43.16         3  2017-03-05
U.S.      2017-03-01  43.16         3  2017-04-05
U.S.      2016-02-01  59.95         4  2016-02-28
U.S.      2016-01-01  99.95         6  2016-01-05
[...]
```

- We can also have multiple tables defined:

```
with
    LTVData as (select
        first_value(locale) over(partition by userid order by trans_dt asc ) as locale
        , date_trunc('month', first_value(trans_dt)
            over(partition by userid order by trans_dt asc))::date as cohort
        , amt, userid, trans_dt
    from
        cls.trans )
    , SubScribersFirst as (select distinct userid from
        (select userid, first_value( type ) over(partition by userid
            ORDER BY trans_dt asc, type asc ) as firsttype
            from cls.trans) as innerQ
        where firsttype = 'Sub')
select
    *
from
    SubScribersFirst
left join
    LTVData
using(userid);


  userid  locale    cohort        amt  trans_dt
--------  --------  ----------  -----  ----------
       2  Canada    2018-08-01  12.99  2018-08-25
       3  U.S.      2017-03-01  43.16  2017-03-05
       3  U.S.      2017-03-01  43.16  2017-04-05
       5  Canada    2018-03-01  17.98  2018-03-09
       5  Canada    2018-03-01  17.98  2018-05-09
[...]
```

- The upside of using a CTE is that they can be much easier to read.
- There are two major downsides to using CTEs:
  1. Some databases do not support them (MySQL)
  2. In other databases they can act as optimization barriers. In particular, consider the following query:

```
with
    LTVData as (select
        first_value(locale)
            over(partition by userid order by trans_dt asc ) as locale
        , date_trunc('month', first_value(trans_dt)
            over(partition by userid order by trans_dt asc)) as cohort
        , amt, userid, trans_dt
    from
        cls.trans )
select
    *
from
    LTVData
where userid = 2;


locale    cohort                       amt    userid  trans_dt
--------  -------------------------  -----  --------  ----------
Canada    2018-08-01 00:00:00+00:00  12.99         2  2018-08-25
```

In this example, it is clear that the filter WHERE userid = 2 could be applied within the LTVData expression. However, it is not and the database will compute the entire LTVData before applying the filter, a costly choice. We will explore performance considerations in the next section.