Chapter 9

Advanced Joins

Contents

1 The Shape of Data **149**

2 Revenue over time & Advanced Joins **151**

2.1 First Value 152

2.2 Most common value by group 156

2.3 Cumulative Sum 158

2.4 Rolling 90 day Calculation 160

2.5 Cohorted Monthly Revenue 161

1 The Shape of Data

- Up to this point we have taken the data given to us as a given: The columns and rows are what they are. However, it is often useful to reshape the data by interchanging rows and columns for other purposes. For example, consider the following two tables:

Table 9.1: Example of wide data: *house_wide*

owner_name	NoBedroomHouse1	NoBedRoomHouse2	CostHouse1	CostHouse2
Rick	3	2	250000	125000
Harry	2	3	250000	125000
James	1		125000	
Lenka	3		450000	

Table 9.2: Example of long data: *house_long*

owner_name	HouseNo	BedRoom	Cost
Rick	1	3	250000
Rick	2	2	125000
Harry	1	2	250000
Harry	2	3	125000
James	1	1	125000
Lenka	1	3	450000

- We would characterize the first table as being "wide" and the second as being "long." While both tables contain the same information depending on the application one shape can be easier to use than the other. Consider the following two questions:

1. What is the average cost of a person's second house?

```
select avg( CostHouse2 ) as avg_cost from cls.house_wide;


  avg_cost
----------
    125000
```

```
select avg(Cost) from cls.house_long where HouseNo = 2;


   avg
------
125000
```

2. What is the average cost of any house?

```
select
    (sum( CostHouse1 ) + sum( CostHouse2 ) )
        / (count( CostHouse1) + count(CostHouse2)) as avg_cost
from cls.house_wide;


  avg_cost
----------
    220833
```

```
select avg(Cost) as avg_cost  from cls.house_long;


  avg_cost
----------
    217500
```

- Looking at the examples above you can see that, even if the case of these simple statistics different data shapes can make a big difference. This is especially important when exporting data to another program.
- We can use GROUP BY and CASE statements to reshape data from long-to-wide:

```
select
    owner_name
    , max( case when HouseNo = 1
        then BedRoom else null end ) as NoBedroomHouse1
    , max( case when HouseNo = 2
        then BedRoom else null end ) as NoBedroomHouse2
    , max( case when HouseNo = 1
        then Cost else null end ) as CostHouse1
    , max( case when HouseNo = 2
        then Cost else null end ) as CostHouse2
from
    cls.house_long
group by 1;


owner_name    nobedroomhouse1    nobedroomhouse2    costhouse1    costhouse2
------------ ------------------ ------------------ ------------  ------------
Rick                        3                  2        230000        125000
Lenka                       3                           450000
James                       1                           125000
Harry                       2                  3        250000        125000
```

- We can use JOIN and UNION ALL to move between wide-to-long:

```
select
    lhs.owner_name
    , lhs.houseNo
    , case
        when houseNo = 1 then nobedroomhouse1
        when houseNo = 2 then nobedroomhouse2
        else null end as nbr
    , case when houseNo = 1 then costhouse1
        when houseNo = 2 then costhouse2
        else null end as ch
from
    (select distinct owner_name, 1 as houseNo from cls.house_wide
        union all
    select distinct owner_name, 2 as houseNo from cls.house_wide) as lhs
LEFT JOIN
    cls.house_wide
using(owner_name)
where case
        when houseNo = 1 then nobedroomhouse1
        when houseNo = 2 then nobedroomhouse2
    else null end  is not null;


owner_name      houseno    nbr      ch
------------  ---------  -----  ------
James                 1      1  125000
Rick                  1      3  250000
Lenka                 1      3  450000
Harry                 1      2  250000
Rick                  2      2  125000
[...]
```

- These constructs – *wide* vs. *long* are important to be able to swap between. Other programming languages often have commands like “pivot”, “reshape”, “rollup” or “crosstab” that generate data in different forms, sometimes with aggregations occurring.

2 Revenue over time & Advanced Joins

- In this section we consider a common application for reshaping data and that is calculating business statistics from transaction data.

- Consider the following dataset which contains information on a business. This contains transaction information where each row represents a particular event. In this case, the event under consideration is the purchase of special soap bars. There are two types of transactions: single bars and double bars while there are two types: “Unit” which represents a one-off transaction and “Sub” which represents a subscription.

- A very common task when analyzing transaction data is understanding the revenue generated by a customer over time. This number (sometimes called LTV or ARPU) is based on “cohorts” of users, or defined groups of users with similar characteristics.

- Using the above data, how would we calculate the average amount spent by each customer?

Figure 9.1: *Trans* table, 1,063,491 rows

orderid	userid	trans	type	locale	trans_dt	units	coupon	months	amt
0	1	Double bar	Unit	U.S.	2016-05-09	2			39.98
1	2	Single bar	Unit	U.S.	2018-07-09	3			35.97
2	2	Single bar	Unit	U.S.	2018-08-25	1			11.99
3	2	Single bar	Unit	U.S.	2018-02-16	1			11.99
4	3	Single bar	Unit	U.S.	2016-02-28	4			47.96
5	4	Double bar	Sub	Canada	2018-03-09	5	25	2	74.96
6	4	Double bar	Sub	Canada	2018-05-09	5	25	2	74.96
7	5	Single bar	Sub	Canada	2016-01-05	4	35	2	31.17
8	6	Double bar	Unit	U.S.	2017-04-13	2			39.98
9	6	Double bar	Unit	U.S.	2016-07-28	4			79.96

```
select
    sum( amt ) / count(distinct userid) as amtPerUser
from cls.trans;



  amtperuser
------------
     69.4199
```

2.1 First Value

- Let's say that we were interested in understanding how relative countries monetized, how would we calculate the amount per user for each country? In other words, if we defined the cohort based on where a user lives, how would the countries compare?

```
select
    locale
    , sum( amt ) / count(distinct userid) as amtPerUser
from
    cls.trans
group by 1;



locale      amtperuser
--------  ------------
Canada         63.2709
Mexico         72.5456
U.S.           62.5774
```

- What happens if a user moves? How is the average amount per country affected if users can move? How should we handle calculating the average amount per user per country? We would probably want to take the first one that a user appears in:

```
select
    new_locale
    , sum( amt ) / count(distinct lhs.userid) as amtPerUser
from
    (select
        min( trans_dt ) as mindt, userid
    from
        cls.trans
    group by 2) as lhs
join
    (select
        userid, locale as new_locale, trans_dt
    from
        cls.trans) as rhs
on
    lhs.mindt = rhs.trans_dt
    and lhs.userid = rhs.userid
left join
    cls.trans
on lhs.userid = trans.userid
group by 1;


new_locale      amtperuser
------------  ------------
Canada             69.3196
Mexico            106.897
U.S.               65.8775
```

Take a look at how the query works. This is an example of identifying a “first value” of a customer. In this case we first identify the column that we are interested in ordering by, identifying the row of interest and then re-joining to the original data based on that row.

- What is the total amount spent by customers by first purchase type (subscription vs. unit sale)? In order to do this we must identify what the first purchase was for each user:

```
select
    lhs.userid, trans.type
from
    (select
        userid, min(trans_dt) as firstdt
    from
        cls.trans
    group by 1) as lhs
left join
    cls.trans
on
    lhs.userid = trans.userid
    and lhs.firstdt = trans.trans_dt


  userid  type
--------  ------
       4  Units
       6  Units
       7  Sub
      10  Sub
      21  Units
[...]
```

What if we a user can make multiple purchases in a day – What do we do in this case? Lets assume that we want to prioritize Subscriptions over Units, so that if a user makes multiple purchases in a day that they are flagged as subscribers:

```
 select
    lhs.userid
    , max( case when trans.type = 'Sub' then 1
        else 0 end ) as subscriber_flag
    , min( firstdt) as firstdt
from
    (select
        userid, min(trans_dt) as firstdt
    from
        cls.trans
    group by 1) as lhs
left join
    cls.trans
on
    lhs.userid = trans.userid
    and lhs.firstdt = trans.trans_dt
group by 1;


  userid    subscriber_flag  firstdt
--------  -----------------  ----------
   84925                  0  2018-05-10
  165533                  0  2018-10-12
  162195                  0  2018-11-14
   47051                  0  2016-03-06
  161180                  1  2016-01-18
[...]
```

- Now that we have identified the type of user, we then need to re-remerge that back onto the data to get the rest of the information that we need:

```
select
    subscriber_flag
    , count(distinct outerLHS.userid) as numusers
    , sum( amt) as totalamt
    , sum(amt) / count(distinct outerLHS.userid) as avg
from
    (select
        lhs.userid
        , max( case when type = 'Sub' then 1
            else 0 end ) as subscriber_flag
        , min( firstdt) as firstdt
    from
        (select
            userid, min(trans_dt) as firstdt
        from
            cls.trans
        group by 1) as lhs
    left join
        cls.trans
    on
        lhs.userid = trans.userid
        and lhs.firstdt = trans.trans_dt
    group by 1) as outerLHS
left join cls.trans
using(userid)
group by 1;


  subscriber_flag    numusers     totalamt      avg
-----------------  ----------  -----------  -------
                0      379309  2.40611e+07  63.434
                1      194980  1.5806e+07   81.0648
```

2.2 Most common value by group

- Another very common task is to find the most common value for a particular group. For example, lets say that we want to figure out what the most common value is among a particular sub group.
- For example, what is the most common order amount (dollars) for each country?

```
select
    locale, amt, count(1)
from
    cls.trans
group by 1,2
order by 3 desc;


locale      amt    count
--------  -----  -------
U.S.      39.98    65523
U.S.      23.98    64606
U.S.      59.97    51018
U.S.      35.97    50809
U.S.      19.99    34005
[...]
```

- Looking at the query above we can see that the most common amount for the US is 39.98, while in Canada and Mexico the amounts are 25.99 and 17.98 respectively. We now want to write a query which identifies just those three values. To do this we need to take this table and join it on itself. Lets look at the following query:

```
select lhs.locale, lhs.amt, lhs.ct
from
    (select locale, amt, count(1) as ct from cls.trans
     group by 1,2) as lhs
left join
    (select locale, amt, count(1) as ct from cls.trans
     group by 1,2) as rhs
on lhs.locale = rhs.locale and lhs.ct <= rhs.ct
group by 1,2,3
having count(rhs.*) = 1;


locale      amt     ct
--------  -----  -----
Canada    25.99  24411
Mexico    35.97  21847
U.S.      39.98  65523
```

- This query works by exploding the dataset via the left join and then collapsing it down along all the left hand side variables. The join itself only matches those counts from the left hand side which are less than or equal to those on the right hand. In other words, this creates a row numbering based on the original count! If you want to see this, run the previous query while removing the final GROUP BY and HAVING.

- This technique can be used to also find the least common value (swapping the inequality to a greater than) or even the second or third highest value (how would this be done?)

2.3 Cumulative Sum

- Another common, difficult query to write is to write a cumulative sum, which adds up all values previous to and including the current row. We need to use the same technique as in the previous examples, but this time use the trans_dt field to help us order the columns:

```
select
    lhs.userid, lhs.amt, lhs.trans_dt
    , sum(rhs.amt) as cumsum
from
    (select userid, amt, trans_dt from cls.trans) as lhs
left join
    (select userid, amt, trans_dt from cls.trans) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt >= rhs.trans_dt
group by 1,2,3
order by 1,3;


  userid    amt  trans_dt       cumsum
--------  -----  ----------  --------
       1  23.98  2016-05-09     23.98
       2  12.99  2018-08-25     12.99
       3  43.16  2017-03-05     43.16
       3  43.16  2017-04-05     86.32
       4  59.95  2016-02-28     59.95
[...]
```

- What if there were multiple values on a particular day?
- In the case of multiple days you the above query will actually generate data since the merge is not unique on each side! This is bad – the sum of the amount of money should be conserved, but if we generate rows the number will actually increase. So how would we get around this? We can sum up by date to make sure that each row is unique by date:

```
select
    lhs.userid, lhs.amt, lhs.trans_dt
    , sum(rhs.amt) as cumsum
from
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as lhs
left join
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt >= rhs.trans_dt
group by 1,2,3
order by 1,3;


  userid    amt  trans_dt      cumsum
--------  -----  ----------  --------
       1  23.98  2016-05-09     23.98
       2  12.99  2018-08-25     12.99
       3  43.16  2017-03-05     43.16
       3  43.16  2017-04-05     86.32
       4  59.95  2016-02-28     59.95
[...]
```

By doing this aggregation we now avoid any creating any data.

- What is we wanted to do the above, but *not* include the current date? To do this we modify the join condition:

```
select
    lhs.userid, lhs.amt, lhs.trans_dt
    , sum(rhs.amt) as cumsum
from
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as lhs
left join
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt > rhs.trans_dt
group by 1,2,3
order by 1,3;


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

2.4 Rolling 90 day Calculation

- What happens when we move into a new locale? If we calculate the average revenue using the ways described above then any new country will look terrible because it is simply younger than the other countries.

- To get rid of this issue we *always* cohort users by when they begin a service. This allows us to compare apples-to-apples, rather than biasing our analysis toward those cohorts which have had more time to matriculate within the system.

- Lets say that we wish to do a rolling calculation – say I want to calculate the average transaction size for the first three months for each customer?

```
select lhs.userid, lhs.trans_dt, lhs.amt, sum( rhs.amt)
from
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as lhs
left join
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt >= rhs.trans_dt
    and lhs.trans_dt <= rhs.trans_dt + 90
group by lhs.userid, lhs.trans_dt, lhs.amt;


  userid  trans_dt      amt    sum
--------  ----------  -----  -----
       1  2016-05-09  23.98  23.98
       2  2018-08-25  12.99  12.99
       3  2017-03-05  43.16  43.16
       3  2017-04-05  43.16  86.32
       4  2016-02-28  59.95  59.95
[...]
```

2.5 Cohorted Monthly Revenue

- For plotting purposes we often want to break down the revenue over time, by the cohort or install date.
- In the following example, we calculate this by month of first transaction and then we return the results in a wide format. Why would we return this data in a wide format? Because this allows us to plot it fairly easily.

```
select
    cohort::date
    , count(distinct userid) as numusers
    , sum(case when trans_dt::date
        between cohort and (cohort + '1 month'::interval)::date
        then amt else 0 end ) as mon_0_amt
    , sum(case when trans_dt::date
        between (cohort + '1 month'::interval)::date
            and (cohort + '2 month'::interval)::date
        then amt else 0 end ) as mon_1_amt
    , sum(case when trans_dt::date
        between (cohort + '2 month'::interval)::date
            and (cohort + '3 month'::interval)::date
        then amt else 0 end ) as mon_2_amt
from
    cls.trans as lhs
left join
    (select userid, date_trunc( 'month', min( trans_dt)) as cohort
    from cls.trans group by 1) as rhs
using(userid)
GROUP BY 1;


cohort        numusers    mon_0_amt    mon_1_amt    mon_2_amt
----------  ----------  -----------  -----------  -----------
2016-01-01       21302       891314      68182         132567
2016-02-01       19503       819087      65729.4       125490
2016-03-01       20339       850657      67757.5       130519
2016-04-01       19571       819085      65591.1       124968
2016-05-01       19408       812544      65243.4       125081
[...]
```

- If we wanted to do this long, we could do the following. Note that by making the data long, we don't need to have an artificial monthly cut-off:

```
select
    rhs.cohort::date
    , rhs2.newusers
    , 12* ( DATE_PART('year', trans_dt::date) - DATE_PART('year', rhs.cohort)  )
    + (DATE_PART('month', trans_dt::date) - DATE_PART('month', rhs.cohort)) as numMonths
    , sum( amt) as revenue
from
    cls.trans as lhs
left join
    (select userid, date_trunc( 'month', min( trans_dt)) as cohort
    from cls.trans group by 1) as rhs
using(userid)
left join
    (select count(distinct userid) as newusers, cohort
    from
        (select userid, date_trunc( 'month', min( trans_dt)) as cohort
        from cls.trans group by 1) as innerrhs
    group by 2) as rhs2
on rhs.cohort = rhs2.cohort
GROUP BY 1,2,3


cohort        newusers    nummonths    revenue
----------  ----------  -----------  ---------
2016-01-01       21302            0   889046
2016-01-01       21302            1    63285.4
2016-01-01       21302            2   131443
2016-01-01       21302            3    27301.3
2016-01-01       21302            4    69052.8
[...]
```

Annoyingly, look at what we had to do to get the number of new users within each cohort into the resulting data!.