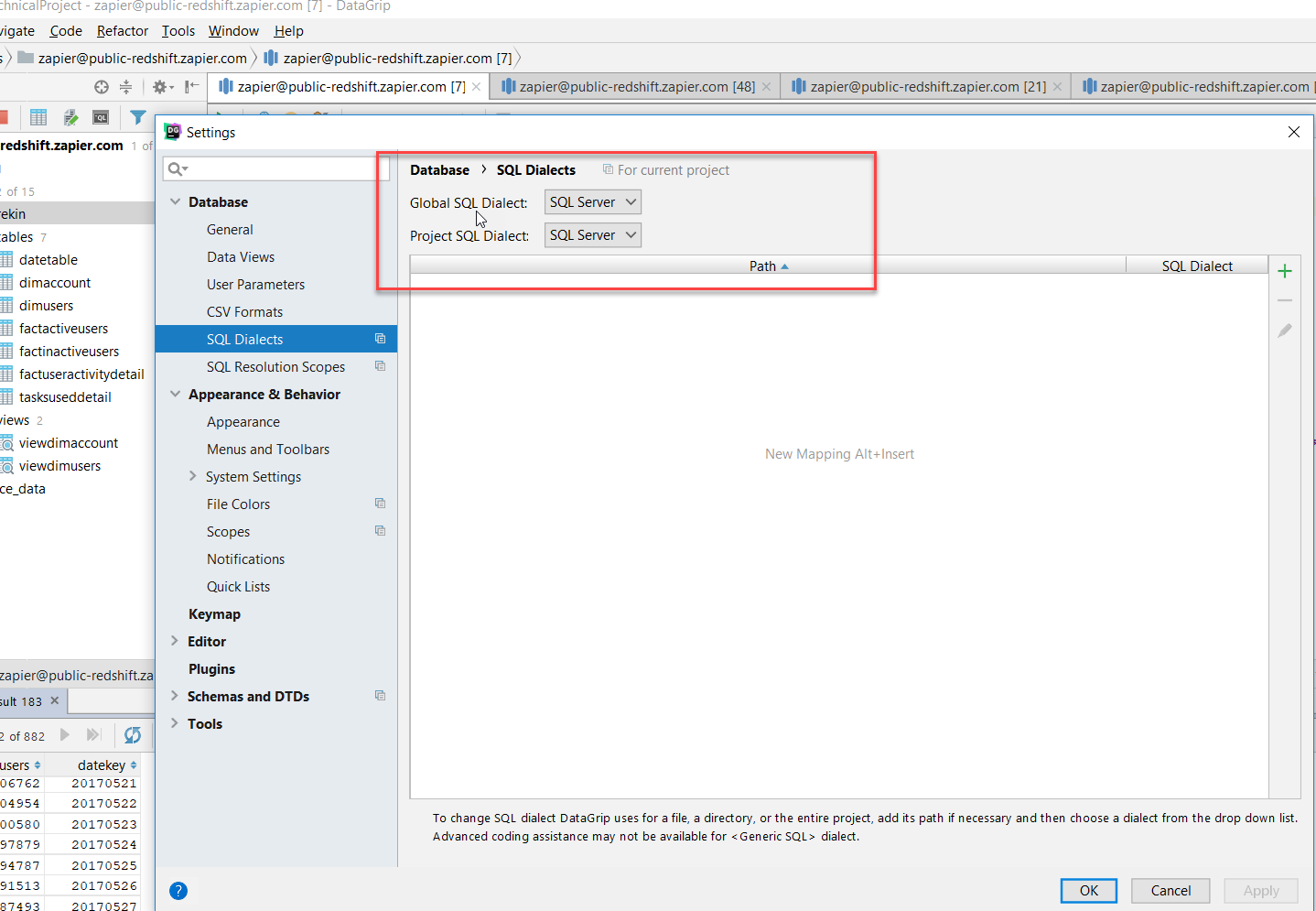
Daily Active Users/Churn

## Overall Methodology

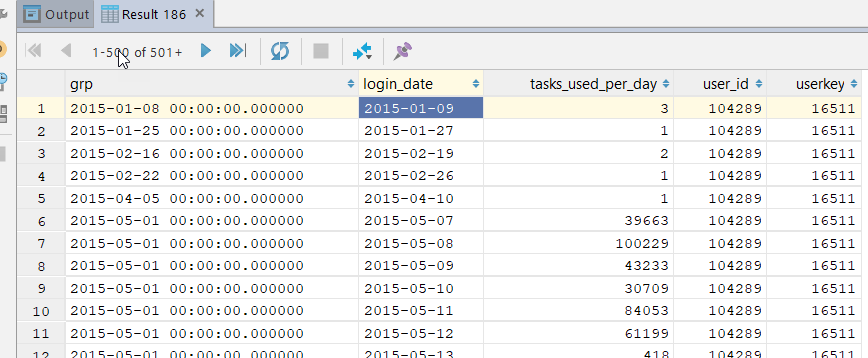
1. **DataGrip is the data management tool chosen to connect to the source schema [source\_data] and to create a dimensional model in [jentrekin] schema.**
2. **DataGrip SQL Dialect is set to use SQL Server**
   1. The author’s experience is working within SQL Server environments, so in order for the scripts that are in the Gist submission to work properly require you to set your dialect to SQL Server.



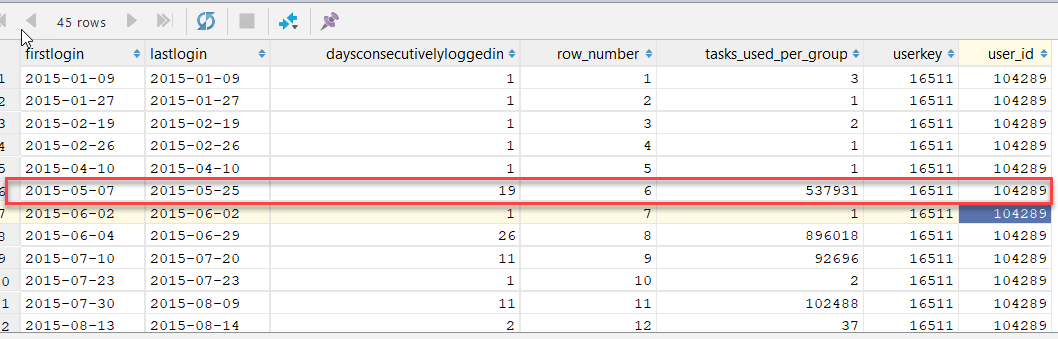
1. **Orchestration** would have used Matillion ETL as this robust product is designed and optimized to be used with Redshift columnstore databases. However, I ran out of time and was not able to orchestrate the jobs. However, this would be a trivial task and I do have a list of execution steps in the README file accompanying this submission.
2. **Split out User/Account to separate dimension tables**
   1. To set up a proper star schema, it was apparent that UserId, UserDescription, Accountid, AccountDescription fields would be necessary, and would also be good candidates for dimensional tables.
   2. Setting up the DDL to use an Identity column to create surrogate keys, allows the fact table to have the same field as foreign key references.
   3. AccountId ---> translates to the dimAccount.AccountKey surrogate key
   4. UserID ----> translates to the dimUsers.UserKey surrogate key
   5. Essentially we are populating a distinct list of uses/accounts in each dimension table.
3. **ViewDimAccount, viewDimUsers**
   1. The views here are required as a limitation of the designer’s knowledge as it pertains to Redshift.
   2. Since Redshift does not allow the use of IDENTITY\_INSERT commands, the view essentially takes care of this by adding a single UNION ALL statement that appends a -1 value for AccountKey/UserKey.
   3. This is for the case where a lookup transformation between a fact record and the dimensional table produces a NULL value – we don’t want NULL foreign keys, so we are assigning an arbitrary value to designate that this is an unknown dimension member for now.
4. **DateTable**
   1. A common strategy is to build out a list of calendar dates for each day of a given year so that analysis such as getting counts by day become much easier.
   2. The DateKey configuration is going to be YYYYMMDD as an integer value. Dates in the transaction/fact tables can easily be converted to this integer representation. Example 2017-05-01 = 20170501
   3. A current day of year flag is also embedded in the data population script. This would allow you to only pull back counts for the ‘current’ day if this was a database that was updated daily.
5. **Nested CTE’s (Common Table Expressions)**
   1. The use of nested CTE’S allows multiple groups of code to reference previously built data sets.
   2. This is similar to temp table methodology, but doesn’t require writing to tempdb on SQL Server, and you can reference complex logic as a single field name instead of having to encapsulate complex transformations inside of other commands
6. **SQL Server Window Functions**
   1. SQL Server window functions have been employed to break up a given user’s range of login activity.
   2. Row\_Number() Over (order by… partition by…) is employed to get either a list of unique row numbers OR a logical partitioning of data that makes certain analysis simpler.
   3. The goal is to break out a set of ‘groupings’ where there are gaps between logins by flattening out the multiple transactions into current and next images. Then we can compare the current login activity to the next login activity, find out how the largest gaps that exist between logins and ultimately determining a set of Type II SCD valid from/valid to date ranges for each set of login groupins.
   4. By using the recursive nature of login\_date – row\_number() over (order by user\_id, login\_date), we can create a ‘grouping’ of dates that correspond to consecutive logins.
7. **Using userid = 104289 as an example, let’s walk through the first section of the nested CTE process:**
   1. Step 1: The first part of the nested CTE process first creates a list of distinct userid’s, sum of tasks\_used, user\_id and UserKey (sk value).
   2. Step2: Using the SQL Server window function login\_date - *row\_number*() **OVER** (**ORDER BY user\_id**, login\_date) **AS** grp, we essentially build out a list of login activity along with a grouping anchor. For example, see below where the login\_date = 2017-05-07. You can see that any consecutive days where there is login activity, the anchor grp value remains constant. This allows us to be able to reduce the number of transactions as well as being able to track the largest gaps in activity.

--STEP 1

**WITH** login\_dates **AS** (  
 **SELECT** *sum*(**tasks\_used**) **as** tasks\_used\_per\_day, **user\_id**, **userkey**,  
 **date as** login\_date  
 **from** jentrekin.tasksuseddetail  
 **where user\_id** = 104289  
 **group by user\_id**, **userkey**,  
 **date  
 order by** login\_date  
 )  
*-- ,--STEP2   
-- login\_date\_groups AS (* **SELECT** login\_date,  
 login\_date - *row\_number*() **OVER** (**ORDER BY user\_id**, login\_date) **AS** grp, -- Subtracts the current row\_number from the current login\_date – if the logins are sequential (back-to-back days), this value is going to remain constant, thus signaling that there were no gaps between logins.  
 tasks\_used\_per\_day,  
 **user\_id**,  
 **userkey  
 FROM** login\_dates

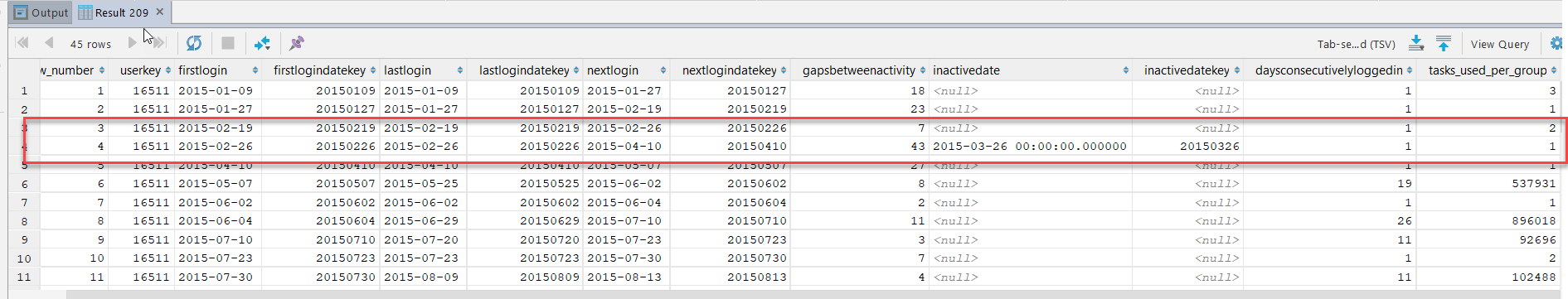


1. Staying with the same userid (104289), let’s continue walking through this example.
   1. In the third layer of the CTE, we’re now defining the ‘first’ login and ‘last’ login for each group from before.
   2. Staying with the 2015-05-01 group, we can calculate that the min login was 2015-05-07 and the max login was 2015-05-25 and thus the days consecutively logged in was 19.
   3. Additionally, the process is defining a unique row number for each user\_id and grp value to make it easier to compare the current last login to the next first login. This will allow us to calculate Churn!



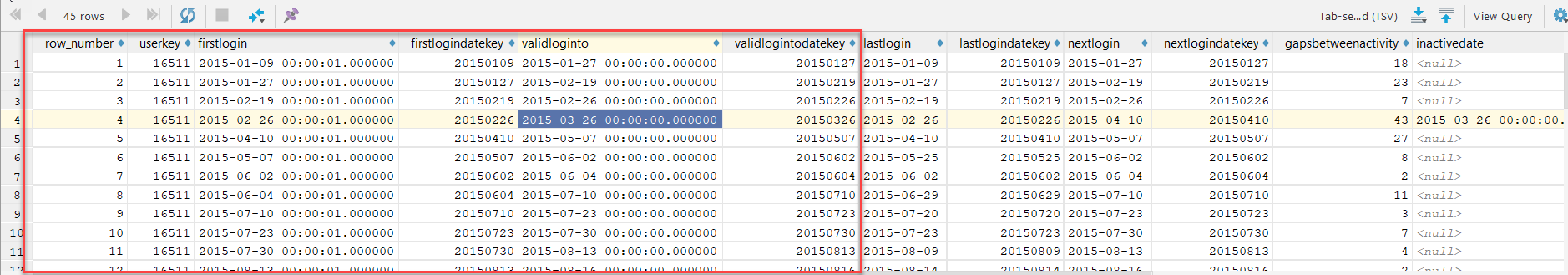
**WITH** login\_dates **AS** (  
 **SELECT** *sum*(**tasks\_used**) **as** tasks\_used\_per\_day, **user\_id**, **userkey**,  
 **date as** login\_date  
 **from** jentrekin.tasksuseddetail  
 **where user\_id** = 104289  
 **group by user\_id**, **userkey**,  
 **date  
 order by** login\_date  
 )  
 ,  
 login\_date\_groups **AS** (  
 **SELECT** login\_date,  
 login\_date - *row\_number*() **OVER** (**ORDER BY user\_id**, login\_date) **AS** grp,  
 tasks\_used\_per\_day,  
 **user\_id**,  
 **userkey  
 FROM** login\_dates  
 )  
 *--,login\_group\_ranges as (* **SELECT** *min*(login\_date) **AS** FirstLogin,  
 *max*(login\_date) LastLogin,  
 *max*(login\_date) - *min*(login\_date) + 1 **AS** [DaysConsecutivelyLoggedIn],  
 *row\_number*()  
 **OVER** (  
 **ORDER BY user\_id**, grp ) **as** row\_number,  
 *sum*(tasks\_used\_per\_day) **as** tasks\_used\_per\_group,  
 **userkey**,  
 **user\_id  
 FROM** login\_date\_groups  
 **GROUP BY** grp, **userkey**, **user\_id**

1. In the 4th level of the nested CTE’s we finally begin to calculate gaps between activity and inactive(churn) dates.
   1. Here we are executing a self-join on the prior piece of the nested CTE on row\_number + 1 = row\_number, as highlighted below. You can see that the row\_number function from before becomes very handy as it’s now a trivial matter to “lookup” the next grouping’s first login.
   2. The self-referencing join allows us to compare the current group of logins to the ‘next’ group of logins. Using date\_diff, we can easily calculate the # of days between the groups. If it’s > 28, the we’re going to calculate an InactiveDate as LastLogin + 28 days.



**WITH** login\_dates **AS** (  
 **SELECT** *sum*(**tasks\_used**) **as** tasks\_used\_per\_day, **user\_id**, **userkey**,  
 **date as** login\_date  
 **from** jentrekin.tasksuseddetail  
 **where user\_id** = 104289  
 **group by user\_id**, **userkey**,  
 **date  
 order by** login\_date  
 )  
 ,  
 login\_date\_groups **AS** (  
 **SELECT** login\_date,  
 login\_date - *row\_number*() **OVER** (**ORDER BY user\_id**, login\_date) **AS** grp,  
 tasks\_used\_per\_day,  
 **user\_id**,  
 **userkey  
 FROM** login\_dates  
 )  
 ,login\_group\_ranges **as** (  
 **SELECT** *min*(login\_date) **AS** FirstLogin,  
 *max*(login\_date) LastLogin,  
 *max*(login\_date) - *min*(login\_date) + 1 **AS** [DaysConsecutivelyLoggedIn],  
 *row\_number*()  
 **OVER** (  
 **ORDER BY user\_id**, grp ) **as** row\_number,  
 *sum*(tasks\_used\_per\_day) **as** tasks\_used\_per\_group,  
 **userkey**,  
 **user\_id  
 FROM** login\_date\_groups  
 **GROUP BY** grp, **userkey**, **user\_id** )  
  
*--,login\_date\_ranges AS (* **select** l.row\_number,  
 l.**userkey**,  
 l.FirstLogin,  
 *cast*(*replace*(*cast*(l.FirstLogin **as varchar**(10)), **'-'**, **''**) **as int**) **as** FirstLoginDateKey,  
 l.LastLogin,  
 *cast*(*replace*(*cast*(l.LastLogin **as varchar**(10)), **'-'**, **''**) **as int**) **as** LastLoginDateKey,  
 l2.FirstLogin **as** NextLogin,  
 *cast*(*replace*(*cast*(l2.FirstLogin **as varchar**(10)), **'-'**, **''**) **as int**) **as** NextLoginDateKey,  
 *cast*(*datediff*(**day**, l.LastLogin, *isnull*(l2.FirstLogin, *getdate*())) **as int**) **as** GapsBetweenActivity,  
 **case when** *datediff*(**day**, l.LastLogin, *isnull*(l2.FirstLogin, *getdate*())) > 28  
 **then** *dateadd*(**day**, 28, l.lastlogin)  
 **else NULL END as** InactiveDate,  
 *cast*(*replace*(*cast*( **case when** *datediff*(**day**, l.LastLogin, *isnull*(l2.FirstLogin, *getdate*())) > 28  
 **then** *dateadd*(**day**, 28, l.lastlogin)  
 **else NULL END as varchar**(10)), **'-'**, **''**) **as int**) **as** InactiveDateKey,  
 L.DaysConsecutivelyLoggedIn,  
 l.tasks\_used\_per\_group  
 **from** login\_group\_ranges l **LEFT JOIN** login\_group\_ranges l2  
 **on** l.row\_number + 1 = l2.row\_number  
 **and** l.**user\_id** = l2.**user\_id  
 order by** l.**userkey**, l.FirstLogin

1. Finally, we’re pulling it all together in a final output where we define the final list of output fields to the detail fact table.
   1. When the InactiveDate from the prior sections is NOT NULL then we’re using that else we’re using the “next” login as the ValidLoginTo datetime field.
   2. If GapsBetweenActivity > 28 THEN we’re assigning a value of 1 for ChurnFlag else 0.



**WITH** login\_dates **AS** (  
 **SELECT** *sum*(**tasks\_used**) **as** tasks\_used\_per\_day, **user\_id**, **userkey**,  
 **date as** login\_date  
 **from** jentrekin.tasksuseddetail  
 **where user\_id** = 104289  
 **group by user\_id**, **userkey**,  
 **date  
 order by** login\_date  
 )  
 ,  
 login\_date\_groups **AS** (  
 **SELECT** login\_date,  
 login\_date - *row\_number*() **OVER** (**ORDER BY user\_id**, login\_date) **AS** grp,  
 tasks\_used\_per\_day,  
 **user\_id**,  
 **userkey  
 FROM** login\_dates  
 )  
 ,login\_group\_ranges **as** (  
 **SELECT** *min*(login\_date) **AS** FirstLogin,  
 *max*(login\_date) LastLogin,  
 *max*(login\_date) - *min*(login\_date) + 1 **AS** [DaysConsecutivelyLoggedIn],  
 *row\_number*()  
 **OVER** (  
 **ORDER BY user\_id**, grp ) **as** row\_number,  
 *sum*(tasks\_used\_per\_day) **as** tasks\_used\_per\_group,  
 **userkey**,  
 **user\_id  
 FROM** login\_date\_groups  
 **GROUP BY** grp, **userkey**, **user\_id** )  
  
,login\_date\_ranges **AS** (  
 **select** l.row\_number,  
 l.**userkey**,  
 l.FirstLogin,  
 *cast*(*replace*(*cast*(l.FirstLogin **as varchar**(10)), **'-'**, **''**) **as int**) **as** FirstLoginDateKey,  
 l.LastLogin,  
 *cast*(*replace*(*cast*(l.LastLogin **as varchar**(10)), **'-'**, **''**) **as int**) **as** LastLoginDateKey,  
 l2.FirstLogin **as** NextLogin,  
 *cast*(*replace*(*cast*(l2.FirstLogin **as varchar**(10)), **'-'**, **''**) **as int**) **as** NextLoginDateKey,  
 *cast*(*datediff*(**day**, l.LastLogin, *isnull*(l2.FirstLogin, *getdate*())) **as int**) **as** GapsBetweenActivity,  
 **case when** *datediff*(**day**, l.LastLogin, *isnull*(l2.FirstLogin, *getdate*())) > 28  
 **then** *dateadd*(**day**, 28, l.lastlogin)  
 **else NULL END as** InactiveDate,  
 *cast*(*replace*(*cast*( **case when** *datediff*(**day**, l.LastLogin, *isnull*(l2.FirstLogin, *getdate*())) > 28  
 **then** *dateadd*(**day**, 28, l.lastlogin)  
 **else NULL END as varchar**(10)), **'-'**, **''**) **as int**) **as** InactiveDateKey,  
 L.DaysConsecutivelyLoggedIn,  
 l.tasks\_used\_per\_group  
 **from** login\_group\_ranges l **LEFT JOIN** login\_group\_ranges l2  
 **on** l.row\_number + 1 = l2.row\_number  
 **and** l.**user\_id** = l2.**user\_id  
 order by** l.**userkey**, l.FirstLogin  
 )  
**SELECT** *row\_number*, **userkey**, *dateadd*(**millisecond**, 1000, FirstLogin) **as** FirstLogin, FirstLoginDateKey, LastLogin, LastLoginDateKey, NextLogin, NextLoginDateKey, GapsBetweenActivity,  
 InactiveDate, InactiveDateKey, DaysConsecutivelyLoggedIn, tasks\_used\_per\_group  
 , **case when** InactiveDate **is not null then** InactiveDate **else** NextLogin **END as** ValidLoginTo  
 , *cast*(*replace*(*cast*(**case when** InactiveDate **is not null then** InactiveDate **else** NextLogin **END as varchar**(10)), **'-'**, **''**) **as int**) **as** ValidLoginToDateKey  
, **case when** GapsBetweenActivity > 28 **then** 1 **else** 0 **end as** ChurnFlag  
**FROM** login\_date\_ranges  
**order by** FirstLogin;

1. At this point, we’re ready to populate our subset fact tables that contain churn transactions and active user transactions by day.
2. Inserting into factInactiveUsers is trivial as we’re just grabbing the records where churnflag = 1

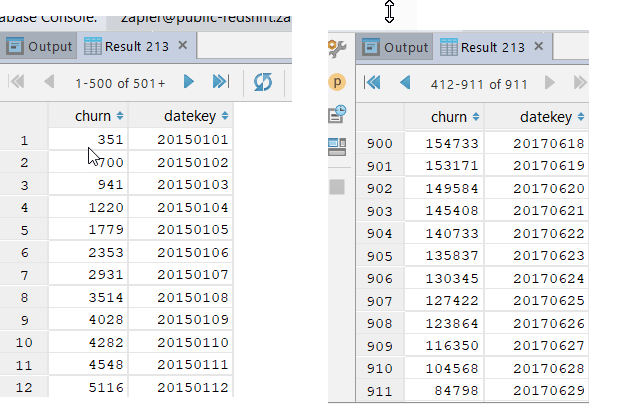
**truncate table** factinactiveusers  
**INSERT INTO** factinactiveusers (**row\_number**, **userkey**, **validfromdate**, **validfromdatekey**, **validtodate**, **validtodatekey**)  
**select** *row\_number*, **userkey**, **lastlogin**, **lastlogindatekey**, **validloginto**, **validlogintodatekey from** factuseractivitydetail  
**where churnflag** = 1

1. Inserting into factActiveUsers simply involves grabbing the transactions where churnflag <> 1

**truncate table** factactiveusers  
**INSERT INTO** factactiveusers (**row\_number**, **userkey**, **validfromdate**, **validfromdatekey**, **validtodate**, **validtodatekey**)  
**select** *row\_number*, **userkey**, **firstlogin**, **firstlogindatekey**, **validloginto**, **validlogintodatekey  
from** factuseractivitydetail  
**where churnflag** <> 1  
**order by firstlogindatekey**

1. Finally, we come to the final steps where we actually calculate churn and active users by day.
   1. Churn :

*--CHURN***select** *count*(**distinct** f.**UserKey**) **as** Churn, dT.**DateKey from** factinactiveusers f  
**JOIN** dimusers dUsers  
 **on** f.**userkey** = dUSers.**userkey  
JOIN** datetable dT  
 **on** dT.**calendar\_date between** f.**validfromdate and** f.**validtodate** *-- and dT.current\_day\_of\_year\_flag = 1  
 --and dUsers.user\_id =104289***group by** dt.**DateKey  
order by** dt.**datekey**;



*--ACTIVE***select** *count*(**distinct** f.**userkey**) **as** ActiveUsers, dT.**DateKey  
from** factActiveUsers f  
 **JOIN** dateTable dT  
 **on** dT.**calendar\_date between** f.**validfromdate and** f.**validtodate  
group by** dt.**DateKey  
order by** dt.**datekey**;

