Aginity SQL Academy

Release 0.5

Jul 03, 2019

SQL Academy

CHAPTER 1

SQL Academy for Data Analysts

1.1 Lesson #1: Calculate Internal Rate of Return (IRR) Directly in Redshift

Internal Rate of Return (IRR) is a calculation frequently used to estimate an investment's rate of return.

I encountered it at a large mortgage lender where IRR was one of many calculations used to determine the health of individual loans. The inputs were the mortgage principle, the previous monthly payments, and estimated monthly payments to the end of the loan period. So, there could easily be more than 300 inputs to the IRR calculation. The standard practice was to transfer a sample of borrower data from the data warehouse to the SAS server to run the IRR calculation and transfer the results back to the data warehouse.

The type of iterative calculation required by IRR is not traditionally done on MPP data warehouse platforms. However, with a bit of advanced SQL thinking, this type of calculation can take advantage of the distributed nature of the MPP to score very large populations.

As with all of our lessons if you want to catalog this code in Aginity Pro or Aginity Team click the

to skip right to the queries to add to your catalog.

1.1.1 Step 1 - Introduction to IRR

$$\text{NPV} = \sum_{0}^{N} \frac{C_n}{(1+r)^n} = 0$$

So, what is the IRR calculation?

Advanced SQL writers come from a range of backgrounds. You may not have a computer science degree, or remember a lot of your high school or college math, so I'll jump into a few details here. The *Additional Resources* section below has links to several useful articles regarding IRR. I'll give a minimal explanation here, so you should read those. The purpose here is to demonstrate some useful SQL techniques. If you need to implement this, or any other financial calculation, in production, then work with an experienced financial engineer.

In the equation above, our goal is to find "r", which is the interest rate that makes the Net Present Value (NPV) equal to 0. In the early 19th century, Evariste Galois proved that this type of polynomial has no general solution; that is, there is no "formula" for finding "r". You have probably encountered these types of functions in your mathematical career. Hopefully, your math teacher discussed a bit about how your fancy scientific calculator, or computer program, was able to solve these equations without a formula.

In Computer Science, the study of finding "good enough" solutions to impossible mathematical problems is called Numerical Methods or Numerical Analysis. For this problem, we are trying to find the "r" that makes NPV equal to zero. So, with our computer program, we can use this method:

```
1. Make a guess for "r"
2. Calculate the NPV based on that guess
3. Evaluate whether or not the NPV resulting from that guess is "close enough" to 0
  a. if it is "close enough" then we are done and our guess is a "good enough" result␣
↪for r
  b. if not, then we need a better guess
4. Improve the guess and start over at #1
```

1.1.2 Step 2 - Relational Database Equivalent of Iteration

It is clear how this type of algorithm can be implemented in a traditional programming language, maybe even in a database system with a procedural language available. However, the power of the MPP lies in SQL, not in the procedure. In order to implement this in SQL, we have to change our thinking from "one at a time" to "all at once". Think of this simple case. In a programming language, you might print the numbers one though 10 like this:

```
for i=1, 10 do
  print(i)
end
```

Assuming that I have a numbers table, in SQL I achieve the same result like this:

```
select num
from numbers
where num <= 10
order by num
;


.. code-block:: aginity_catalog_item
```

In the procedural language, it is clear that the computer "did" something (printed 1), then "did" something else (printed 2). Did this happen in the database system? Maybe, maybe not. The point is that SQL is focused on the **result set**. The **SET** is only in a particular order because I asked for it to be displayed that way. There is a **SET** of numbers. I don't care about the order in which they were read off the disk. Let's go a step further:

```
total = 0
for i=1, 10 do
  total = total + i
end
print(total)
```

vs

```
select sum(num) as total
from
```

(continues on next page)

(continued from previous page)

```
    (select num
     from numbers
     where num <= 10
     ) as inner_select
;
```

Again, the *order* of the process described by the procedural language matches what the system actually does. Does the database system keep a running total when it calculates a SUM? I don't know. It doesn't matter. The SQL says that, of the SET returned by the inner_select, give me the sum. When there is a logical ordering of operations – in order to SUM over a set I first have to identify a set – then that order of operations is clear in the SQL.

So, how does this apply to our problem of making guesses? In procedural languages, we make one guess at a time, evaluate it, and make another guess; in SQL we can make *all of our guesses at the same time*.

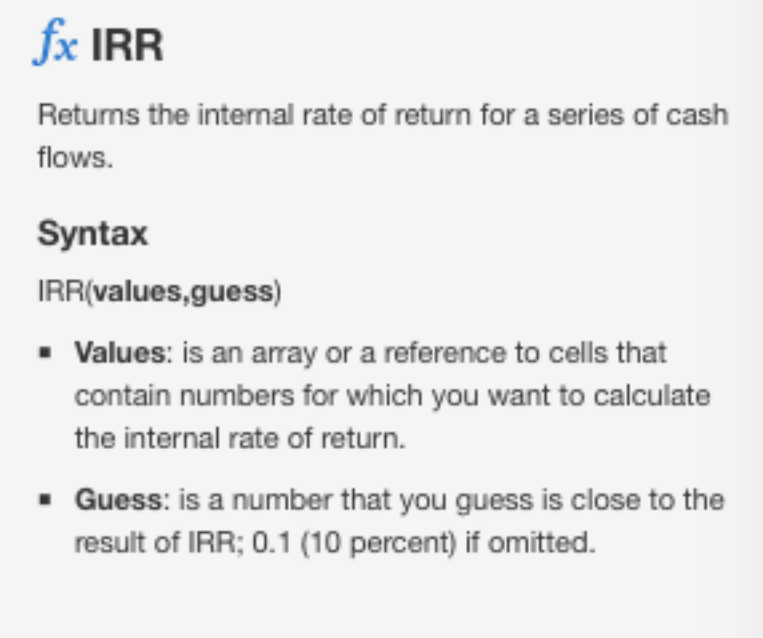
1.1.3 Step 3 - IRR Baby Steps

In breaking down a calculation like this, I like to start with all the terms on a single row and work backward from there. We can easily get as many examples as we want by using the IRR calculation in Excel. So, let's look a simple excel example:

Table 1: Excel Example 1

Period	Payment	IRR Excel
0	-10,000	
1	2,400	
2	2,400	
3	2,400	
4	2,400	
5	2,400	
6	2,400	11.530473216%

Note: IRR Excel Function



fx IRR

Returns the internal rate of return for a series of cash flows.

Syntax

IRR(**values,guess**)

- **Values**: is an array or a reference to cells that contain numbers for which you want to calculate the internal rate of return.
- **Guess**: is a number that you guess is close to the result of IRR; 0.1 (10 percent) if omitted.

Remember that anything raised to the power of "0" is equal to 1. So, we can use a "0" period to get the value of the payment "as-is".

If "Net Present Value" is the sum of a bunch of things, let's call those pre-summed things "Present Value". So, for every payment period we need:

- an ID so we can handle multiple borrowers
- the period number
- the payment for that period

```
-- example 1
drop table if exists payment_data;
create temp table payment_data
(id int, period int, payment int)
;

insert into payment_data VALUES
(1, 0, -10000), (1, 1, 2400), (1, 2, 2400), (1, 3, 2400), (1, 4, 2400), (1, 5, 2400),␣
↪(1, 6, 2400)
;

-- section 1
select *
from payment_data
order by id, period
;
```

Output | Result 1

id	period	payment
1	0	-10000
1	1	2400
1	2	2400
1	3	2400
1	4	2400
1	5	2400
1	6	2400

Let's build up the calculation. Remember, we are trying to get an NPV of 0. For our example data, Excel has told us that happens when r is about "0.115305".

$$\text{NPV} = \sum_{0}^{N} \frac{C_n}{(1+r)^n} = 0$$

1. for a given period, the numerator is the payment for that period
2. inside the parentheses, is (1 + 0.115305)
3. the dominator is the parens raised to the power of the period: **power((1 + 0.115305), period)**

So, the query is

```
select id, period, payment, payment/(power(1+0.115305, period)) as pv
from payment_data
order by id, period
;
```

Let's sum that to see whether it is correct:

```
select sum(pv) as npv from
  (select id, period, payment, payment/(power(1+0.115305, period)) as pv
   from payment_data
   order by id, period
  ) as inner_query
;
```

Output | Result 1

npv
-0.007646283151188982

So, that's pretty close to 0. Theoretically, we can get as close to 0 as we want by continually adding decimal places to our value of r.

In this case, we "cheated" by getting Excel to tell us the correct value for now. Next we are going to evaluate r over a range of "guesses" to determine which value of r produces an NPV close enough to 0.

Sidebar - Numbers

Note: A note about the *numbers* table

As we've tried to emphasize, SQL is driven by sets. We have a set of payments; we need a set of "guesses". Needing a set of numbers is so common in SQL that many database systems have a function to generate numbers as needed. For example, PostgreSQL 9 has *generate_series()*. Unfortunately, Redshift and many other MPPs lack this feature; fortunately in MPPs, we have lots of things we can count in order to generate numbers.

In Redshift, the **stl_plan_info** table has a record for every query you have run. If you have been using Redshift for a while, then this may be sufficient. Otherwise a cross join will give you plenty. Generate a numbers table with one million rows.

```
drop table if exists numbers;
create temp table numbers as
select num from
(select cast(row_number() over (partition by 1) as int) as num from stl_plan_info
 -- uncomment the next line if you need more data
 -- cross join stl_plan_info b
) inner_query
where num <= 1000000
;
```

1.1.4 Step 4 - Longer Strides

So, we know from Excel that the IRR for our data is between 11% and 12%. Let's explore a few values.

First a table of guesses:

```
drop table if exists guesses;
create temp table guesses as
select num*.01 as guess
from numbers
;
```

In the previous example, we had the "guess" hard coded. Now we want our guesses table to drive the guess. So, every row of our payment data needs its own guess. In SQL, we can achieve this by using **cross join**. In SQL development we always need to keep in mind an estimate of the sizes of our input and output sets so things don't get out of hand. We have 7 periods; we'll look at 10 guesses initially. That will be 70 inner rows that will aggregate to 10 npv_guesses to evaluate. Rather than make this strictly true by pre-limiting our guesses table, we'll assume that Redshift is smart enough to do that limitation for us. If not, then we'll have 7 million inner rows that will be filtered down to 70. For Redshift, we won't consider that to be a big deal for now.

So our inner query is this:

```
select id, period, payment, guess, payment/(power(1+guess, period)) as pv
from payment_data
cross join guesses
```

(continues on next page)

(continued from previous page)

```
where guess between 0.06 and .15
order by id, period, guess
```

We can't tell much by looking at this level of detail, so let's aggregate

```
select id, guess, sum(pv) as npv_guess
from
  (select id, period, payment, guess, payment/(power(1+guess, period)) as pv
   from payment_data
   cross join guesses
   where guess between 0.06 and .15  -- an arbitrary limit on guess for easy viewing
   order by id, period, guess
  ) as inner_query
group by id, guess
order by id, guess
;
```

Output | Result 1

id	guess	npv_guess
1	0.06	1801.578382412936
1	0.07	1439.6951834338545
1	0.08	1094.911193506856
1	0.09	766.204616554236
1	0.1	452.6256787093389
1	0.11	153.29084897181383
1	0.12	-132.62242354641967
1	0.13	-405.8805064579683
1	0.14	-667.1979602979764
1	0.15	-917.2415345848985

We can see that the value closest to 0 is .12. Let's dial in additional precision by adding decimals to our guesses, then re-running the aggregate query:

```
drop table if exists guesses;
create temp table guesses as
select num*.001 as guess
from numbers
;
```

Run the npv_guess query again.

Output | Result 1

id	guess	npv_guess
1	0.114	37.359079273394855
1	0.115	8.705679597800554
1	0.116	-19.817448205043547
1	0.117	-48.211067273444314

Now there are 100 rows of output and closest to 0 is .115. Let's jump a couple levels of precision and re-run the aggregate query.

```
drop table if exists guesses;
create temp table guesses as
select num*.00001 as guess
from numbers
;
```

Output | Result 1

id	guess	npv_guess
1	0.11528	0.7061053285563048
1	0.11529	0.42059493025544725
1	0.1153	0.13509753690505022
1	0.11531	-0.15038685225431436
1	0.11532	-0.43585823798571255
1	0.11533	-0.7213166210469808

Now we have 10,000 rows with the closest being 0.11530.

Note: If you are working through these queries, go ahead and put the decimal back to "0.01" in the guesses table so we get faster execution times for the rest of the examples.

1.1.5 Step 5 - Crossing the Finish Line

Now we can see the shape of where we are going. We are making all of our guesses "at the same time"; at least as part of the same result set. From that set, we need to find the one that is closest to 0; that is, the npv_guess that has the minimum absolute value.

As our levels of aggregation continue to grow, we need to be comfortable with the technique of SQL Window functions.

Let's rewrite the previous query with a window function. Also, remember the previous note to take the guesses table back down to two decimal places for faster execution.

```
select *, payment/(power(1+guess, period)) as pv,
       sum(payment/(power(1+guess, period))) over(partition by id, guess order by␣
↪period rows unbounded preceding) as npv_guess,
       max(period) over(partition by id, guess) as max_period
from payment_data
cross join guesses
order by id, guess, period
```

Now we have our payment data, pv, and npv_guess on the same row. The npv_guess aggregation is being driven by the sum using the window function. For aggregating at the next level, where we are going to find the npv_guess closest to 0, we need to choose a row. The row we want is the one with the last period for our data. So, we have a max_period aggregation that we'll use for a filter at the next level. Note that the final "order by" clause here and in examples below is for us to use in visualizing the output. The aggregation here is based on the "order by" clause inside the window function.

Any time we use window functions, we want to add test cases to make sure that the functions are working as expected. So, let's add a couple of more IRR examples from excel:

Table 2: Excel Example 2

Period	Payment	IRR Excel
0	-1,000	
1	120	
2	360	
3	100	
4	240	
5	480	8%

Table 3: Excel Example 3

Period	Payment	IRR Excel
0	-18,000	
1	3,100	
2	2,400	
3	2,400	
4	2,400	
5	2,400	
6	2,400	
7	3,000	
8	3,200	
9	3.600	7%

```
insert into payment_data VALUES
(2, 0, -1000), (2, 1, 120), (2, 2, 360), (2, 3, 100), (2, 4, 240), (2, 5, 480),
(3, 0, -18000), (3, 1, 3100), (3, 2, 2400), (3, 3, 2400), (3, 4, 2400), (3, 5, 2400),␣
↪(3, 6, 2400),
(3, 7, 3000), (3, 8, 3200), (3, 9, 3600)
;
```

At this level, we have all of our guesses, along with their distances from 0 (absolute value), and identification of which of these is the closest to 0.

```
select id, guess, abs(npv_guess) as abs_npv_guess,
       min(abs(npv_guess)) over(partition by id) as min_abs_npv_guess
```

(continues on next page)

(continued from previous page)

```
from
 (select *, payment/(power(1+guess, period)) as pv,
      sum(payment/(power(1+guess, period))) over(partition by id, guess order by␣
↪period rows unbounded preceding) as npv_guess,
      max(period) over(partition by id, guess) as max_period
  from payment_data
  cross join guesses
  order by id, guess, period
 ) as payment_level
where period = max_period
order by id
;
```

So, one additional filter gives the final query:

```
select id, guess as irr
from
  (select id, guess, abs(npv_guess) as abs_npv_guess,
       min(abs(npv_guess)) over(partition by id) as min_abs_npv_guess
   from
     (select *, payment/(power(1+guess, period)) as pv,
          sum(payment/(power(1+guess, period))) over(partition by id, guess order by␣
↪period rows unbounded preceding) as npv_guess,
          max(period) over(partition by id, guess) as max_period
      from payment_data
      cross join guesses
      order by id, guess, period
     ) as payment_level
   where period = max_period
   order by id, guess
     ) as guess_level
 where abs_npv_guess = min_abs_npv_guess
 order by id
;
```

Output | Result 1

id	guess
1	0.12
2	0.08
3	0.07

1.1.6 IRR Sample Aginity Catalog Assets

1.1.7 Additional Resources

https://www.mathsisfun.com/money/internal-rate-return.html

https://exceljet.net/excel-functions/excel-irr-function

https://medium.com/@_orcaman/package-financial-for-golang-the-math-behind-the-irr-function-1eedf225d9f

https://www.codeproject.com/Tips/461049/Internal-Rate-of-Return-IRR-Calculation

http://ci.columbia.edu/ci/premba_test/c0332/s5/s5_5.html

https://en.wikipedia.org/wiki/Internal_rate_of_return

https://www.investopedia.com/terms/i/irr.asp

1.2 Lesson #5: Learn to Create Basic Recency, Frequency and Monetary Segments

RFM (Recency, Frequency, Monetary) analysis is a simple, easy to comprehend, highly used marketing model for behavior based customer segmentation. It groups customers based on their transaction history – how recently did they transact, how often did they transact and how much did they purchase.

RFM models helps divide customers into various categories or clusters to identify customers who are more likely to respond to promotions and also for future personalization services.

See also:

As with all of our lessons if you want to catalog this code in Aginity Pro or Aginity Team click the

to skip right to the queries to add to your catalog.

1.2.1 Step 1 - The Required Data

For the sake of this example we want to make it simple but let you expand it against any data you have available to you.

We are going to mock up some data using a basic SQL **UNION ALL** command. We will use two different tables which are very typical in RFM modeling, **TRANSACTION_HEADER** and **TRANSACTION_DETAIL**.

Here is the structure of each table.

TRANSACTION_HEADER

Column name	Column Description	Data Type
customer_id	This is a unique identifier of a customer that purchased.	Integer
transaction_id	A unique identifier of a transaction.	Integer
transaction_date	The date on which the transaction occurred.	Date

TRANSACTION_DETAIL

Column name	Column Description	Data Type
transaction_id	A unique identifier of a transaction. FK to TRANSACTION HEADER.	Integer
quantity	The quantity of items purchased.	Integer
net_amount	The total amount of items purchased.	Decimal(14,3)

Sample RFM Data Scripts

The following SQL Scripts are portable and used to simulate the RFM model.

TRANSACTION_HEADER

```
-- transaction_header data

select cast(123456 as integer) as customer_id, cast(11111 as integer) as transaction_id, cast('2019-01-01' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11112 as integer) as transaction_id, cast('2019-01-04' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11113 as integer) as transaction_id, cast('2019-01-07' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11114 as integer) as transaction_id, cast('2019-01-10' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11115 as integer) as transaction_id, cast('2019-01-14' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11116 as integer) as transaction_id, cast('2019-01-17' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11117 as integer) as transaction_id, cast('2019-01-20' as date) as transaction_date
union all
select cast(123456 as integer) as customer_id, cast(11118 as integer) as transaction_id, cast('2019-01-27' as date) as transaction_date
union all
select cast(234567 as integer) as customer_id, cast(21115 as integer) as transaction_id, cast('2019-01-14' as date) as transaction_date
union all
select cast(234567 as integer) as customer_id, cast(21116 as integer) as transaction_id, cast('2019-01-15' as date) as transaction_date
union all
select cast(234567 as integer) as customer_id, cast(21117 as integer) as transaction_id, cast('2019-01-16' as date) as transaction_date
union all
select cast(234567 as integer) as customer_id, cast(21118 as integer) as transaction_id, cast('2019-01-17' as date) as transaction_date
```

TRANSACTION_DETAIL

```
    --transaction_detail data

```

(continues on next page)

(continued from previous page)

```
    select cast(11111 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(11112 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(11112 as integer) as transaction_id, cast(1 as integer) as quantity, cast(7.25 as decimal(13,2)) as net_amount
    union all
    select cast(11113 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(11112 as integer) as transaction_id, cast(15 as integer) as quantity, cast(1.10 as decimal(13,2)) as net_amount
    union all
    select cast(11114 as integer) as transaction_id, cast(1 as integer) as quantity, cast(25.34 as decimal(13,2)) as net_amount
    union all
    select cast(11114 as integer) as transaction_id, cast(2 as integer) as quantity, cast(14.32 as decimal(13,2)) as net_amount
    union all
    select cast(11114 as integer) as transaction_id, cast(1 as integer) as quantity, cast(7.10 as decimal(13,2)) as net_amount
    union all
    select cast(11115 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(11116 as integer) as transaction_id, cast(1 as integer) as quantity, cast(8.10 as decimal(13,2)) as net_amount
    union all
    select cast(11117 as integer) as transaction_id, cast(2 as integer) as quantity, cast(23.10 as decimal(13,2)) as net_amount
    union all
    select cast(11118 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(21115 as integer) as transaction_id, cast(14 as integer) as quantity, cast(4.10 as decimal(13,2)) as net_amount
    union all
    select cast(21116 as integer) as transaction_id, cast(16 as integer) as quantity, cast(8.10 as decimal(13,2)) as net_amount
    union all
    select cast(21117 as integer) as transaction_id, cast(4 as integer) as quantity, cast(23.10 as decimal(13,2)) as net_amount
    union all
    select cast(21118 as integer) as transaction_id, cast(1 as integer) as quantity, cast(43.10 as decimal(13,2)) as net_amount
```

1.2.2 Step 2 - The RFM Query

We are using a **WITH** statement to collapse two passes into one required SQL statement. Let's start with the statement itself that returns the aggregated analytics and the segment each customer falls in.

```
with customer_metrics as
(
```

(continues on next page)

(continued from previous page)

```
    select
     th.customer_id,
     count(distinct th.transaction_id) as trips_per_period,
     sum(td.quantity * td.net_amount) as total_spend_per_period,
     datediff(DAY,current_date, max(th.transaction_date)) AS days_since_last_
↪transaction
    from  TRANSACTION_HEADER th
    join  TRANSACTION_DETAIL td on th.transaction_id = td.transaction_id
    where th.transaction_date > dateadd(day, cast ($lookback_days as integer)*-1,
↪current_date)
    AND td.quantity > 0             -- returns ignored
    group by th.customer_id),
    rfm as
    (
    select customer_id,
      ntile($buckets) over (order by days_since_last_transaction desc) as r,
      ntile($buckets) over (order by trips_per_period desc) as f,
      ntile($buckets) over (order by total_spend_per_period desc) as m,
      trips_per_period,
      total_spend_per_period,
      days_since_last_transaction
    from customer_metrics
    )
    select customer_id
         , r
         , f
         , m
         , trips_per_period
         , total_spend_per_period
         , days_since_last_transaction
         , ntile($buckets) over (order by 1.0*r+1.0*f+1.0*m)
    from rfm;  -- weights on final calculation - default to 1.0
```

Breakdown of the SQL

There are two queries embedded in the **WITH** statement: **customer_metrics** and **rfm**.

The **customer_metrics** query will aggregate:

1. The trips_per_period by counting distinct transaction id's (F or Frequency).
2. The total_spend_per_period by summing net_sales and the quantity sold (M or Monetary).
3. The days_since_last_transaction by finding the difference between the current date and the last purchase date (R or Recency).

In this query we have two parameters: $lookback_days which tells you how long from current date do you want to segment customers purchases by and $buckets which signifies the number of segments you want the query to return.

The **rfm** query then uses the windowing function, ntile, which will take the ordered data from the **customer_metrics** query and segment them into equal size (number of rows per group).

The final query as shown below brings together all the information from the **WITH** queries and displays it along with a final ntile of the RFM calculation.

```
    select customer_id
          , r
          , f
          , m
          , trips_per_period
          , total_spend_per_period
          , days_since_last_transaction
          , ntile($buckets) over (order by 1.0*r+1.0*f+1.0*m)
     from rfm;  -- weights on final calculation - default to 1.0
```

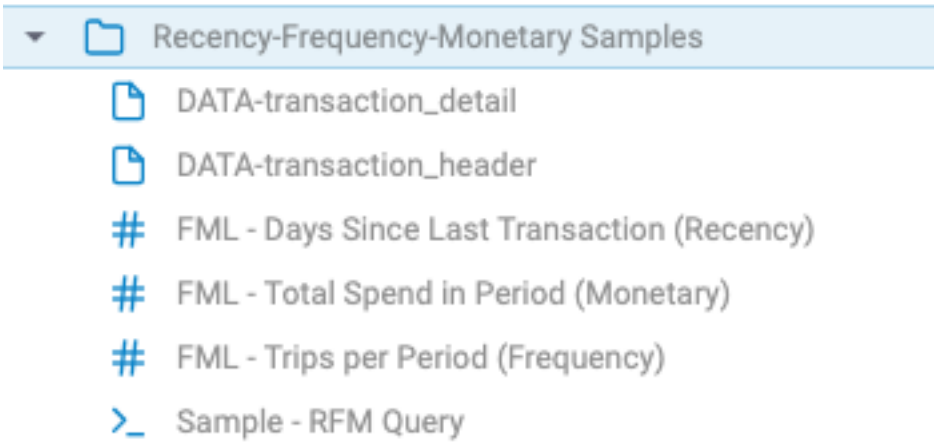
Note: The 1.0 you see in the query above represents equal weight to the R, F and M calculation. In some cases an organization may want to weight each measure differently to perform the final segmentation. For instance, you may choose to apply a weight of 2 to monetary and 1 and 1 to frequency and recency.

Ignore this answer, it does not work: Better use the answer from Louis

For anchor, you may define "short" anchor names like this:

1.2.3 RFM Sample Aginity Catalog Assets

There are six assets you can add to your catalog. I chose to add them as shown below.



These queries are written using ANSII standard SQL so should work across most database platforms. Just select a connection in the Pro/Team Editor and either double click the catalog item and execute or drag and drop the catalog item which will expose the code and run them.

DATA-transaction_header

```
    (select cast(123456 as integer) as customer_id, cast(11111 as integer)␣
↪as transaction_id, cast('2019-01-01' as date) as transaction_date
    union all
    select cast(123456 as integer) as customer_id, cast(11112 as integer) as␣
↪transaction_id, cast('2019-01-04' as date) as transaction_date
    union all
```

(continues on next page)

(continued from previous page)

```
    select cast(123456 as integer) as customer_id, cast(11113 as integer) as transaction_id, cast('2019-01-07' as date) as transaction_date
    union all
    select cast(123456 as integer) as customer_id, cast(11114 as integer) as transaction_id, cast('2019-01-10' as date) as transaction_date
    union all
    select cast(123456 as integer) as customer_id, cast(11115 as integer) as transaction_id, cast('2019-01-14' as date) as transaction_date
    union all
    select cast(123456 as integer) as customer_id, cast(11116 as integer) as transaction_id, cast('2019-01-17' as date) as transaction_date
    union all
    select cast(123456 as integer) as customer_id, cast(11117 as integer) as transaction_id, cast('2019-01-20' as date) as transaction_date
    union all
    select cast(123456 as integer) as customer_id, cast(11118 as integer) as transaction_id, cast('2019-01-27' as date) as transaction_date
    union all
    select cast(234567 as integer) as customer_id, cast(21115 as integer) as transaction_id, cast('2019-01-14' as date) as transaction_date
    union all
    select cast(234567 as integer) as customer_id, cast(21116 as integer) as transaction_id, cast('2019-01-15' as date) as transaction_date
    union all
    select cast(234567 as integer) as customer_id, cast(21117 as integer) as transaction_id, cast('2019-01-16' as date) as transaction_date
    union all
    select cast(234567 as integer) as customer_id, cast(21118 as integer) as transaction_id, cast('2019-01-17' as date) as transaction_date)
```

DATA-transaction_detail

```
      (select cast(11111 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
      union all
      select cast(11112 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
      union all
      select cast(11112 as integer) as transaction_id, cast(1 as integer) as quantity, cast(7.25 as decimal(13,2)) as net_amount
      union all
      select cast(11113 as integer) as transaction_id, cast(3 as integer) as quantity, cast(3.10 as decimal(13,2)) as net_amount
      union all
      select cast(11112 as integer) as transaction_id, cast(15 as integer) as quantity, cast(1.10 as decimal(13,2)) as net_amount
      union all
      select cast(11114 as integer) as transaction_id, cast(1 as integer) as quantity, cast(25.34 as decimal(13,2)) as net_amount
      union all
      select cast(11114 as integer) as transaction_id, cast(2 as integer) as quantity, cast(14.32 as decimal(13,2)) as net_amount
      union all
      select cast(11114 as integer) as transaction_id, cast(1 as integer) as quantity, cast(7.10 as decimal(13,2)) as net_amount
```

(continues on next page)

(continued from previous page)

```
    union all
    select cast(11115 as integer) as transaction_id, cast(3 as integer) as quantity,
↪ cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(11116 as integer) as transaction_id, cast(1 as integer) as quantity,
↪ cast(8.10 as decimal(13,2)) as net_amount
    union all
    select cast(11117 as integer) as transaction_id, cast(2 as integer) as quantity,
↪ cast(23.10 as decimal(13,2)) as net_amount
    union all
    select cast(11118 as integer) as transaction_id, cast(3 as integer) as quantity,
↪ cast(3.10 as decimal(13,2)) as net_amount
    union all
    select cast(21115 as integer) as transaction_id, cast(14 as integer) as␣
↪quantity, cast(4.10 as decimal(13,2)) as net_amount
    union all
    select cast(21116 as integer) as transaction_id, cast(16 as integer) as␣
↪quantity, cast(8.10 as decimal(13,2)) as net_amount
    union all
    select cast(21117 as integer) as transaction_id, cast(4 as integer) as quantity,
↪ cast(23.10 as decimal(13,2)) as net_amount
    union all
    select cast(21118 as integer) as transaction_id, cast(1 as integer) as quantity,
↪ cast(43.10 as decimal(13,2)) as net_amount)
```

FML - Days Since Last Transaction (Recency)

This asset is a reusable formula that calculates the days between execution run time (current date) and the maximum transaction date for each customer.

```
current_date - max(th.transaction_date)
```

FML - Total Spend in Period (Monetary)

This asset is a reusable formula that calculates the aggregation (sum) of net sales, defined as quantity multiplied by net_amount over the specified time period.

```
sum(td.quantity * td.net_amount)
```

FML - Trips per Period (Frequency)

This asset is a reusable formula that counts the number of distinct transactions within the specified time period.

```
  count(distinct th.transaction_id)
```

Sample - RFM Query

This asset uses the formulas above and then calculates the segementation using the windowed analytic function ntile.

```
      with customer_metrics as
      (
      select
      th.customer_id,
      @{/Samples/Sample Data Science Queries - All Platforms/Recency-Frequency-Monetary Samples/FML - Trips per Period (Frequency)} as trips_per_period,
      @{/Samples/Sample Data Science Queries - All Platforms/Recency-Frequency-Monetary Samples/FML - Total Spend in Period (Monetary)} as total_spend_per_period,
      @{/Samples/Sample Data Science Queries - All Platforms/Recency-Frequency-Monetary Samples/FML - Days Since Last Transaction (Recency)} AS days_since_last_transaction
      from  @{/Samples/Sample Data Science Queries - All Platforms/Recency-Frequency-Monetary Samples/DATA-transaction_header}
      join  (SELECT * FROM @{/Samples/Sample Data Science Queries - All Platforms/Recency-Frequency-Monetary Samples/DATA-transaction_detail}) td
      on th.transaction_id = td.transaction_id
      where td.quantity > 0             -- returns ignored
      --and th.transaction_date > dateadd(day, cast($lookback_days as integer)*-1, current_date)  -- consider the past 365 days - customization opportunity (1)
      group by th.customer_id),
      rfm as
      (
      select customer_id,
      ntile($buckets) over (order by days_since_last_transaction desc) as r,  -- split into 10 bins - customization opportunity (2)
      ntile($buckets) over (order by trips_per_period desc) as f,
      ntile($buckets) over (order by total_spend_per_period desc) as m,
      trips_per_period,
      total_spend_per_period,
      days_since_last_transaction
      from customer_metrics
      )
      select customer_id
         , r
         , f
         , m
         , trips_per_period
         , total_spend_per_period
         , days_since_last_transaction
         , ntile($buckets) over (order by 1.0*r+1.0*f+1.0*m)
      from rfm
```

1.3 Lesson #6: Identify Trends Using Linear Regression

It's often helpful to analyze or group customers depending on how their usage (or purchase) of your product is trending. This can be useful for marketing purposes or to identify those likely to attrit. You could do this by comparing their usage last week with their usage this week but often that can be misleading. Possibly they took a vacation last week but that doesn't mean they stopped using your product. If you look at their usage over a longer period of time then a week of zero sales is going to have minimal impact on the overall tread.

1.3.1 Step 1 - Understanding Regression

One approach to doing this is to use Simple Linear Regression. Essentially what we do is to plot the usage over a period of time and then draw a best fit line through those points. This gives us 2 things which are helpful from a

marketing and reporting perspective:-

1. The direction of the trend (whether its up or down) will tell us if usage is increasing or decreasing
2. The slope of the trend (how steep it is) gives us an indication of how quickly it's changing.

This is best shown with an example. The general equation is shown below. In order to get the best fit line 2 constants are calculated:-

- α This is the intercept. It has minimal value for us and we will not be covering it here.
- β This gives us the direction and slope of the trend discussed above.

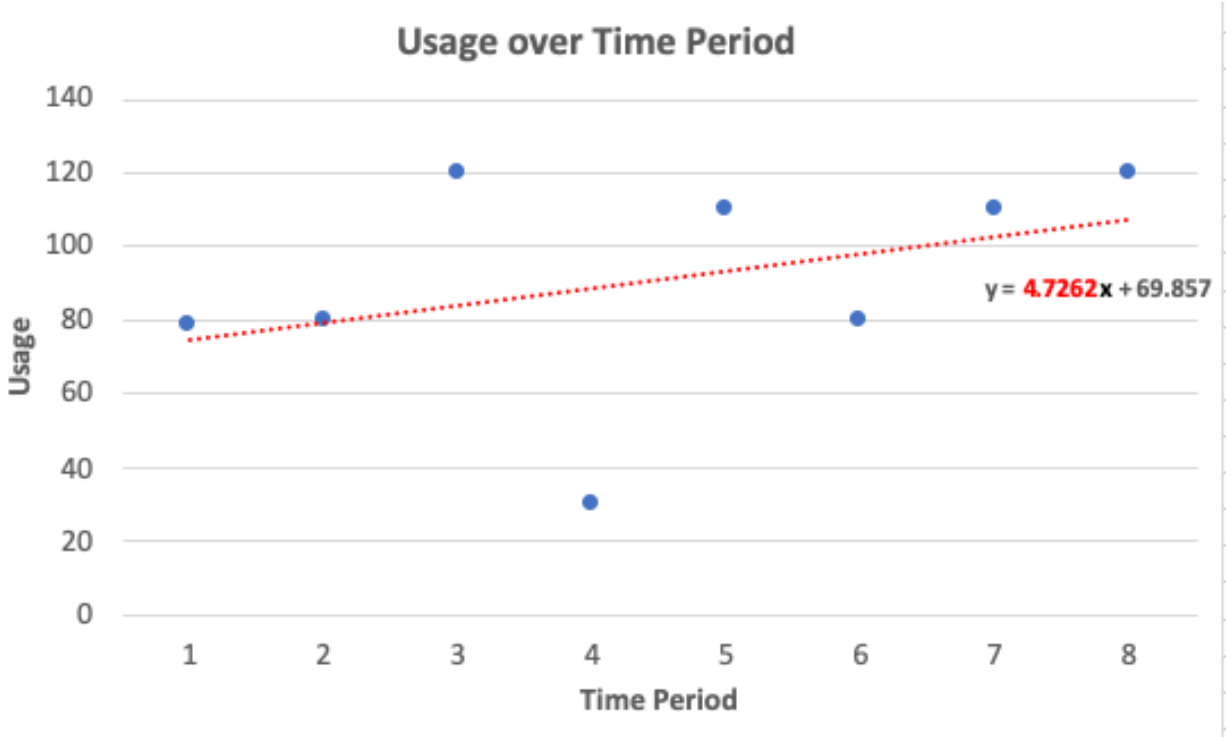
$$y = \alpha + \beta x$$

In our example *x* is the Time Period and *y* is the Usage during that time period.

In the example below we are tracking a specific customer's usage of our product over 8 time periods.

Time Period	Usage
1	79
2	80
3	120
4	30
5	110
6	80
7	110
8	120

Lets look at it graphically.



So you can see from the diagram that the trend is up. What the algorithm has done here is to plot the best fit line through the points we have provided and from that we can easily see the trend. We can also see the the value of β in

the equation is about 4.73. This means that usage roughly increases 4.73 on average over each time period. If you are interested in the math behind all this you can find more details here - Wikipedia Simple Linear Regression.

1.3.2 Step 2 - Building it in SQL

How do we do this in SQL ? It's actually quite simple. Assume we have a simple table called usage with 3 columns

usage

Column name	Column Description	Data Type
customer_id	Unique identifier of a customer	Integer
time_period	Time Period identifier	Integer
usage_cnt	# of times customer used product in time period	Integer

Some sample data is shown below. In this data customer_id 1 is the same as the example discussed above. You can see there are 2 customers each with 8 time periods.

```
--usage data

select 1 as customer_id, 1 as time_period, 79 as usage_cnt union all
select 1 as customer_id, 2 as time_period, 80 as usage_cnt union all
select 1 as customer_id, 3 as time_period, 120 as usage_cnt union all
select 1 as customer_id, 4 as time_period, 30 as usage_cnt union all
select 1 as customer_id, 5 as time_period, 110 as usage_cnt union all
select 1 as customer_id, 6 as time_period, 80 as usage_cnt union all
select 1 as customer_id, 7 as time_period, 110 as usage_cnt union all
select 1 as customer_id, 8 as time_period, 120 as usage_cnt union all
select 2 as customer_id, 1 as time_period, 80 as usage_cnt union all
select 2 as customer_id, 2 as time_period, 70 as usage_cnt union all
select 2 as customer_id, 3 as time_period, 100 as usage_cnt union all
select 2 as customer_id, 4 as time_period, 60 as usage_cnt union all
select 2 as customer_id, 5 as time_period, 40 as usage_cnt union all
select 2 as customer_id, 6 as time_period, 50 as usage_cnt union all
select 2 as customer_id, 7 as time_period, 30 as usage_cnt union all
select 2 as customer_id, 8 as time_period, 40 as usage_cnt
```

The SQL used to get the slope (β) and the results is shown below

```
with prep as (
select customer_id,
sum(time_period)/count(*)  as x_bar,
sum(usage_cnt)/count(*)  as y_bar
from usage group by customer_id
)
select t1.customer_id,
       round(sum((time_period - x_bar)*(usage_cnt - y_bar)) /
             sum((time_period - x_bar)*(time_period - x_bar)), 2) as slope
from usage t1 inner join prep t2 on t1.customer_id = t2.customer_id
group by t1.customer_id
```

The SQL does some prep work to get averages for the the x and y values (x being the Time Period and y being the Usage). Then it applies the linear regression formula to get the slope (β) for each customer. As you can see from the results below Customer 1 has a β of 4.73. The second customer has a negative β which means their usage is decreasing. Using this data you could, for example, assign each customer to a different campaign. e.g. If a customer

has a negative β then they might receive a special offer to extend their subscription. Whereas customers with positive β could be the target of a cross-sell campaign to generate more revenue since they seem to enjoy using the product.

Output | Result 1

t1.customer_id	slope
1	4.73
2	-7.74

1.3.3 Step 3 - Taking it to the Next Step

So far so good. However, the above example assumes that your data is provided in a very convenient way. In practice there are 2 scenarios that may add complication and require additional processing.

(1) **Actual time periods**. It's unlikely that you'll be given numeric periods starting at 1. In most cases you'll be looking at specific time periods - e.g. weeks or months. If the numeric representation of these dates creates consistent spacing between them - e.g. 201801, 201802, 201803 then you are good (there is a spacing of 1 between all these). For the purposes of calculating β we don't care if the x values start at 1 or 201801. However if there is not even spacing between them e.g. 201811, 201812, **201901** then you need to make adjustments or else the slope of your line will be misleading.

(2) **Missing records**. Depending on how your database is set up a missing time period might be automatically created with a value of zero. However in practice it's more likely to missing. If a customer did not use your product for a given time period then there is probably not an entry for it in the database. If this is the case then you'll need to insert default records with a usage of 0 prior to doing your regression. If not your trend will be misleading since it will miss periods where no activity occurs.

Let's take a look at each of these in turn.

Actual time periods

Typically your data will be delivered with a specific time period. In the example data below we have year_month (rather than a simple Time Period like what we had in the first example). You can see below that we jump from 201812 to 201901. If we use the raw year_month value in our regression calculation then the slope will be misleading.

```
1     --usage_month data
2
3     select 1 as customer_id, 201811 as year_month, 79 as usage_cnt union all
4     select 1 as customer_id, 201812 as year_month, 80 as usage_cnt union all
5     select 1 as customer_id, 201901 as year_month, 120 as usage_cnt union all
6     select 1 as customer_id, 201902 as year_month, 30 as usage_cnt union all
7     select 1 as customer_id, 201903 as year_month, 110 as usage_cnt union all
8     select 1 as customer_id, 201904 as year_month, 80 as usage_cnt union all
9     select 1 as customer_id, 201905 as year_month, 110 as usage_cnt union all
10    select 1 as customer_id, 201906 as year_month, 120 as usage_cnt union all
11    select 2 as customer_id, 201811 as year_month, 80 as usage_cnt union all
12    select 2 as customer_id, 201812 as year_month, 70 as usage_cnt union all
13    select 2 as customer_id, 201901 as year_month, 100 as usage_cnt union all
14    select 2 as customer_id, 201902 as year_month, 60 as usage_cnt union all
15    select 2 as customer_id, 201903 as year_month, 40 as usage_cnt union all
16    select 2 as customer_id, 201904 as year_month, 50 as usage_cnt union all
```

(continues on next page)

(continued from previous page)

```
      select 2 as customer_id, 201905 as year_month, 30 as usage_cnt union all
      select 2 as customer_id, 201906 as year_month, 40 as usage_cnt
```

Ideally we want to convert our year_month to a sequential time period like what we had in the first example. The easiest way to do that is to change your Date dimension so that sequential integers are assigned to months when building the Date dimension. Then you simply use that integer field instead of the year_month field. Assume your date dimensions looks something like this (grossly simplified I know :) . You will note that the time_period field does not begin with 1. That is to show that it does not need to. As long as there is an equal interval of 1 between each period then it doesn't matter where I start. We will end up with the same result for β .

```
      --date_dim data

      select 201811 as year_month, 100 as time_period union all
      select 201812 as year_month, 101 as time_period union all
      select 201901 as year_month, 102 as time_period union all
      select 201902 as year_month, 103 as time_period union all
      select 201903 as year_month, 104 as time_period union all
      select 201904 as year_month, 105 as time_period union all
      select 201905 as year_month, 106 as time_period union all
      select 201906 as year_month, 107
```

The following SQL is almost identical to the that shown earlier. It simply has extra joins to the date dimension on year_month to return the sequential time_period. This will give the same result as the prior query.

```
    with prep as
    (   select customer_id,
        sum(time_period)/count(*)  as x_bar,
        sum(usage_cnt)/count(*)  as y_bar
        from usage_month t1 join date_dim t2 on t1.year_month = t2.year_month group␣
↪by customer_id
    )
    select t1.customer_id,
           round(sum((time_period - x_bar)*(usage_cnt - y_bar)) /
                 sum((time_period - x_bar)*(time_period - x_bar)), 2) as slope
    from usage_month t1 join prep t2 on t1.customer_id = t2.customer_id
                    join date_dim t3 on t1.year_month = t3.year_month
    group by t1.customer_id
```

Missing records

In the dataset below some data is missing. e.g. for Customer 1 we are missing data for February (201902) and March (201903). This means they had no usage in these months so for the purposes of calculating the regression we want to assume that usage in these months was zero.

```
--usage_month_with_gaps data

select 1 as customer_id, 201811 as year_month, 79 as usage_cnt union all
select 1 as customer_id, 201812 as year_month, 80 as usage_cnt union all
select 1 as customer_id, 201901 as year_month, 120 as usage_cnt union all
select 1 as customer_id, 201904 as year_month, 80 as usage_cnt union all
select 1 as customer_id, 201905 as year_month, 110 as usage_cnt union all
select 1 as customer_id, 201906 as year_month, 120 as usage_cnt union all
select 2 as customer_id, 201811 as year_month, 80 as usage_cnt union all
select 2 as customer_id, 201812 as year_month, 70 as usage_cnt union all
select 2 as customer_id, 201901 as year_month, 100 as usage_cnt union all
select 2 as customer_id, 201902 as year_month, 60 as usage_cnt union all
```

(continues on next page)

(continued from previous page)

```
select 2 as customer_id, 201905 as year_month, 30 as usage_cnt union all
select 2 as customer_id, 201906 as year_month, 40 as usage_cnt
```

To do this we add yet another step to the SQL. This time we look at all the possible combinations of customer / year_month. Then we create an entry for customer / year_month combinations that are not already in the usage_month_with_gaps table and give those new records a usage_cnt = 0. In the example I'm doing a cross join of all my distinct customers with all the possible year_month values from my date dimension. In practice you'd want to filter this as much as possible for performance reasons.

```
    with usage_prep as
    (   select * from usage_month_with_gaps union all
        select customer_id, year_month, 0 from
        (select customer_id from usage_month_with_gaps t1 group by customer_id) t1
          cross join (select year_month from date_dim) t2
        where not exists ( select 1 from usage_month_with_gaps t3
        where t1.customer_id = t3.customer_id
        and t2.year_month = t3.year_month ) order by customer_id, year_month
    ),
    prep as
    (   select customer_id,
        sum(time_period)/count(*)  as x_bar,
        sum(usage_cnt)/count(*)  as y_bar
        from usage_prep t1 join date_dim t2 on t1.year_month = t2.year_month
        group by customer_id
    )
    select t1.customer_id,
          round(sum((time_period - x_bar)*(usage_cnt - y_bar)) /
                sum((time_period - x_bar)*(time_period - x_bar)), 2) as slope
    from usage_prep t1   join prep t2 on t1.customer_id = t2.customer_id
                         join date_dim t3 on t1.year_month = t3.year_month
    group by t1.customer_id;
```

so the overall trend has not changed but the values of β have changed to account for the zero usage periods.

Output | Result 1

t1.customer_id	slope
1	3.77
2	-10

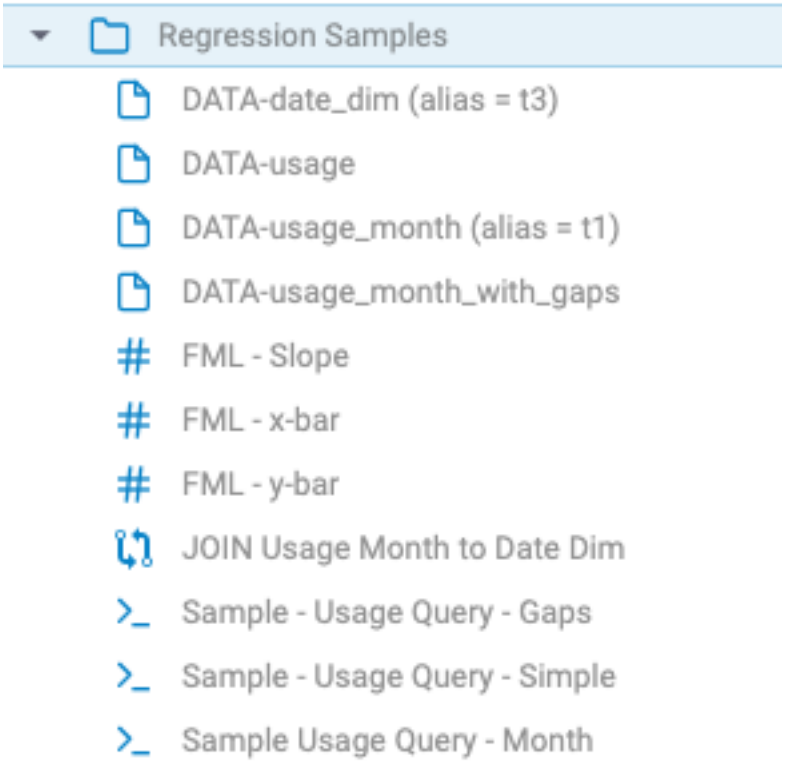
1.3.4 Other things to consider

- Don't choose too many time periods. It obviously depends on your business but recency is typically very important. Do you really care what your customers did a year ago ? You may want to consider a long-term trend (using the regression method we've defined here) in combination with a short-term trend which just uses the last couple of time periods
- In addition to segmenting and marketing based on the direction of the trend also consider the slope - the actual value of β. You may want to consider differentiating between customers with slow growth in usage and those

with quicker growth - similarly where product usage is declining. Remember the β number is the average increase or decrease over the period being analyzed.

1.3.5 Sample Regression Aginity Catalog Assets

There are eleven assets you can add to your catalog. I chose to add them as shown below.



These queries are written using ANSII standard SQL so should work across most database platforms. Just select a connection in the Pro/Team Editor and either double click the catalog item and execute or drag and drop the catalog item which will expose the code and run them.

DATA-date_dim (alias = t3)

This asset is a data set built using union all to represent a typical date dimension. Note in the catalog I add the alias t3. You can choose to do this several ways but this can be handy when using in a **Relationship** catalog item.

```
(select 201811 as year_month, 100 as time_period union all
    select 201812 as year_month, 101 as time_period union all
    select 201901 as year_month, 102 as time_period union all
    select 201902 as year_month, 103 as time_period union all
    select 201903 as year_month, 104 as time_period union all
    select 201904 as year_month, 105 as time_period union all
    select 201905 as year_month, 106 as time_period union all
    select 201906 as year_month, 107
    ) t3
```

DATA-usage

This asset is a data asset that shows a customer, a time period and some "activity" we are labeling usage which we will trend using the regression equation.

```
  (
    select 1 as customer_id, 1 as time_period, 79.0 as usage_cnt union all
          select 1 as customer_id, 2 as time_period, 80.0 as usage_cnt union␣
↪all
          select 1 as customer_id, 3 as time_period, 120.0 as usage_cnt␣
↪union all
          select 1 as customer_id, 4 as time_period, 30.0 as usage_cnt union␣
↪all
          select 1 as customer_id, 5 as time_period, 110.0 as usage_cnt␣
↪union all
          select 1 as customer_id, 6 as time_period, 80.0 as usage_cnt union␣
↪all
          select 1 as customer_id, 7 as time_period, 110.0 as usage_cnt␣
↪union all
          select 1 as customer_id, 8 as time_period, 120.0 as usage_cnt␣
↪union all
          select 2 as customer_id, 1 as time_period, 80.0 as usage_cnt union␣
↪all
          select 2 as customer_id, 2 as time_period, 70.0 as usage_cnt union␣
↪all
          select 2 as customer_id, 3 as time_period, 100.0 as usage_cnt␣
↪union all
          select 2 as customer_id, 4 as time_period, 60.0 as usage_cnt union␣
↪all
          select 2 as customer_id, 5 as time_period, 40.0 as usage_cnt union␣
↪all
          select 2 as customer_id, 6 as time_period, 50.0 as usage_cnt union␣
↪all
          select 2 as customer_id, 7 as time_period, 30.0 as usage_cnt union␣
↪all
          select 2 as customer_id, 8 as time_period, 40.0 as usage_cnt
  )
```

DATA-usage_month (alias = t1)

This asset is a data asset that shows a customer, a monthly time period and some "activity" we are labeling usage which we will trend using the regression equation. Again we alias this catalog item with "t1".

```
(select 1 as customer_id, 201811 as year_month, 79.0 as usage_cnt union all
  select 1 as customer_id, 201812 as year_month, 80.0 as usage_cnt union all
  select 1 as customer_id, 201901 as year_month, 120.0 as usage_cnt union all
  select 1 as customer_id, 201902 as year_month, 30.0 as usage_cnt union all
  select 1 as customer_id, 201903 as year_month, 110.0 as usage_cnt union all
  select 1 as customer_id, 201904 as year_month, 80.0 as usage_cnt union all
  select 1 as customer_id, 201905 as year_month, 110.0 as usage_cnt union all
  select 1 as customer_id, 201906 as year_month, 120.0 as usage_cnt union all
  select 2 as customer_id, 201811 as year_month, 80.0 as usage_cnt union all
  select 2 as customer_id, 201812 as year_month, 70.0 as usage_cnt union all
  select 2 as customer_id, 201901 as year_month, 100.0 as usage_cnt union all
  select 2 as customer_id, 201902 as year_month, 60.0 as usage_cnt union all
  select 2 as customer_id, 201903 as year_month, 40.0 as usage_cnt union all
```

(continues on next page)

(continued from previous page)

```
  select 2 as customer_id, 201904 as year_month, 50.0 as usage_cnt union all
  select 2 as customer_id, 201905 as year_month, 30.0 as usage_cnt union all
  select 2 as customer_id, 201906 as year_month, 40.0 as usage_cnt) t1
```

DATA-usage_month_with_gaps

This asset is a data asset that shows a customer, a monthly time period not sequential and some "activity" we are labeling usage which we will trend using the regression equation.

```
  (
  select 1 as customer_id, 201811 as year_month, 79.0 as usage_cnt union all
  select 1 as customer_id, 201812 as year_month, 80.0 as usage_cnt union all
  select 1 as customer_id, 201901 as year_month, 120.0 as usage_cnt union all
  select 1 as customer_id, 201904 as year_month, 80.0 as usage_cnt union all
  select 1 as customer_id, 201905 as year_month, 110.0 as usage_cnt union all
  select 1 as customer_id, 201906 as year_month, 120.0 as usage_cnt union all
  select 2 as customer_id, 201811 as year_month, 80.0 as usage_cnt union all
  select 2 as customer_id, 201812 as year_month, 70.0 as usage_cnt union all
  select 2 as customer_id, 201901 as year_month, 100.0 as usage_cnt union all
  select 2 as customer_id, 201902 as year_month, 60.0 as usage_cnt union all
  select 2 as customer_id, 201905 as year_month, 30.0 as usage_cnt union all
  select 2 as customer_id, 201906 as year_month, 40.0 as usage_cnt
  )
```

FML - Slope

This formulaic asset is used to store the equation for slope once but reuse in several other regression analysis.

```
sum((time_period - x_bar)*(usage_cnt - y_bar)) /
             sum((time_period - x_bar)*(time_period - x_bar))
```

FML - x-bar

This formulaic asset is used to plot the x axis variable which is the time period in this case.

```
sum(time_period)/count(*)
```

FML - y-bar

This formulaic asset is used to plot the y axis variable which is the average "activity" or usage count in this case.

```
sum(usage_cnt)/count(*)
```

JOIN Usage Month to Date Dim

This is a relationship object that is used to store join paths once and allow you to reuse them.

```
t1.year_month = t3.year_month
```

Note: See the usage of the alias we reference above.

Sample - Usage Query - Simple

This query asset will use the data, formulas and joins described above to perform the regression.

```
 with prep as (
       select customer_id,
            @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/FML - x-bar} as x_bar,
            @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/FML - y-bar} as y_bar
       from @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/DATA-usage} usage
       group by
           customer_id
       )
       select
            t1.customer_id,
            round(@{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/FML - Slope}, 2) as slope
       from @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/DATA-usage} t1
         inner join prep t2 on t1.customer_id = t2.customer_id
       group by
              t1.customer_id
```

Sample - Usage Query - Gaps

This query asset will use the data, formulas and joins described above to fill in gaps with zero data and then perform the regression.

```
  with usage_prep as
       (   select * from @{Samples/Sample Data Science Queries - All Platforms/Regression Samples/DATA-usage_month_with_gaps} usage union all
           select customer_id, year_month, 0 from
           (select customer_id from @{Samples/Sample Data Science Queries - All Platforms/Regression Samples/DATA-usage_month_with_gaps} t1 group by customer_id) t1
            cross join (select year_month from @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/DATA-date_dim (alias = t3)}) t2
           where not exists ( select 1 from @{Samples/Sample Data Science Queries - All Platforms/Regression Samples/DATA-usage_month_with_gaps} t3
           where t1.customer_id = t3.customer_id
           and t2.year_month = t3.year_month )
           order by customer_id, year_month
       ),
       prep as
       (   select customer_id,
           @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/FML - x-bar} as x_bar,
           @{/Samples/Sample Data Science Queries - All Platforms/Regression Samples/FML - y-bar}  as y_bar
```

(continues on next page)

(continued from previous page)

```
            from usage_prep t1 join @{/Samples/Sample Data Science Queries -␣
↪All Platforms/Regression Samples/DATA-date_dim (alias = t3)} on t1.year_
↪month = t3.year_month
            group by customer_id
        )
        select t1.customer_id,
              round(@{/Samples/Sample Data Science Queries - All Platforms/
↪Regression Samples/FML - Slope}, 2) as slope
        from usage_prep t1   join prep t2 on t1.customer_id = t2.customer_id
                             join @{/Samples/Sample Data Science Queries -␣
↪All Platforms/Regression Samples/DATA-date_dim (alias = t3)} on t1.year_
↪month = t3.year_month
        group by t1.customer_id
```

Sample Usage Query - Month

This query asset will use the data, formulas and joins described above aggregate to a month and then perform the regression.

```
 with prep as
 (   select customer_id,
     @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/FML - x-bar}  as x_bar,
     @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/FML - y-bar}  as y_bar
     from @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/DATA-usage_month (alias = t1)}
     join @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/DATA-date_dim (alias = t3)}
       on @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/JOIN Usage Month to Date Dim}
     group by customer_id
 )
 select t1.customer_id,
        round(@{/Samples/Sample Data Science Queries - All Platforms/
↪Regression Samples/FML - Slope}, 2) as slope
 from @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/DATA-usage_month (alias = t1)}
 join prep t2
   on t1.customer_id = t2.customer_id
 join @{/Samples/Sample Data Science Queries - All Platforms/Regression␣
↪Samples/DATA-date_dim (alias = t3)}
        on @{/Samples/Sample Data Science Queries - All Platforms/
↪Regression Samples/JOIN Usage Month to Date Dim}
 group by t1.customer_id
```

CHAPTER 2

SQL Academy for Data Engineers

2.1 Lesson #2: Essential Redshift Utilities: Generate DDL and Search Table Metadata

A decade ago, technologists tended to specialize in a few systems. In the database realm, you might have focused on Oracle or DB2 or SQL Server. Even with massively parallel processing (MPP) databases we tended to specialize in Teradata or Netezza or Greenplum. However, over the past few years, I have worked on projects on all of these systems and more, including cloud-based systems like Hive, Spark, Redshift, Snowflake, and BigQuery. When I worked only in Oracle and only used an Oracle SQL editor, then I knew exactly where to find my store of SQL snippets for doing things like querying the database system tables.

However, as the number of databases I worked with each day expanded, so did the number of places I had to look for my old code, and the amount of time I had to spend researching the intricacies of system tables for those databases. In the well-known Workbench series of products from Aginity, each product focused on one database platform (Workbench for Netezza, Workbench for Redshift, etc.). Aginity Pro is more than a unified platform for executing SQL against many databases. Now there is an "Active Catalog" where I can store, organize, manage, and execute my hard-won SQL.

As SQL analysts and engineers, there are a few things we do every time we approach a new set of data or a new project. Many of those things involve querying the system tables of the current database. The Active Catalog included with Aginity Pro includes a "Sample Catalog" directory with, among other things, system table queries that are ready to use out of the box. Below, I'll discuss two of these. Neither is a particularly sophisticated use of the Active Catalog, but being familiar with their location and usage means that new projects can start immediately with data investigation rather than with a Google search for how to query the system tables in [your database platform here].

2.1.1 Step 1 - Search Table Metadata

Gathering all of the bits and pieces that make up a DDL statement required a fairly large "system" query. Searching for column names *should* be simple. In fact, in most systems it *is* simple after you finish searching Google for that right tables to query. I have researched how to search partial column names on probably 30 database systems over the years. When I start a new project, I frequently get vague instructions like, "you should find what you need in the Current Customer table". When I finally get a database connection, I see table names like FRDST_CST_MN. Huh?

They are all very straight-forward queries. For example, here is the definition of Search for Columns by partial name - Public schema:

```
1  select table_catalog as database, table_schema, table_name, column_name,␣
   ↪ordinal_position as column_sequence
2  from information_schema.columns
3  where lower(table_schema) = 'public'
4  and lower(column_name) like $partial_column_name
```

Note: The $partial_column_name is a parameter and is usually used with % wildcard characters.

2.1.2 Step 2 - Generate Drop Table Query

In some cases you can string together SQL statements to get more value from them. For instance in a lot of cases we desire to search the database catalog for table names that match a pattern and then generate a **DROP** statement to clean the database up. The first query below will search for all tables in the information schema that match a name sequence.

```
1  select table_catalog as database, table_schema, table_name
2  from information_schema.tables
3  where lower(table_name) like lower($partial_table_name);
```

You can now use this SQL and embed it inside of another SQL statement to generate the **DROP** statements

```
1  select 'drop table ' + table_name + ';'
2  from
3    (
4      select table_catalog as database, table_schema, table_name
5      from information_schema.tables
6      where lower(table_name) like lower($partial_table_name)
7    ) a;
```

When executed for tables that match the name '%pendo%' we will return the following results:

```
drop table pendo_featureevents;
drop table pendo_visitor;
drop table pendo_trackevents;
```

2.1.3 Step 3 - Generate DDL

When we sit down to a new project, we frequently need to work with the Data Definition Language (DDL). In most database systems, the actual structure of database objects is spread across several tables. Table-level properties are one place, columns another place, constraints another. Some systems provide a view to pull all of these sources together so that we can easily query the DDL of an existing table. Redshift does not provide a built-in view for this, but Amazon has provided an example query on Github.

This 230 lines of SQL provided by Amazon allows an admin to create a view that can then be queried to assemble the DDL.

```
1      (
2      SELECT
3      table_id
```

(continues on next page)

(continued from previous page)

```
    ,schemaname
    ,tablename
    ,seq
    ,ddl
    FROM
    (
    SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,2 AS seq
    ,'CREATE TABLE IF NOT EXISTS ' + QUOTE_IDENT(n.nspname) + '.' +␣
↪QUOTE_IDENT(c.relname) + '' AS ddl
    FROM pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    WHERE c.relkind = 'r'
    --OPEN PARENT COLUMN LIST
    UNION SELECT c.oid::bigint as table_id,n.nspname AS schemaname,␣
↪c.relname AS tablename, 5 AS seq, '(' AS ddl
    FROM pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    WHERE c.relkind = 'r'
    --COLUMN LIST
    UNION SELECT
    table_id
    ,schemaname
    ,tablename
    ,seq
    ,'\t' + col_delim + col_name + ' ' + col_datatype + ' ' + col_
↪nullable + ' ' + col_default + ' ' + col_encoding AS ddl
    FROM
    (
    SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,100000000 + a.attnum AS seq
    ,CASE WHEN a.attnum > 1 THEN ',' ELSE '' END AS col_delim
    ,QUOTE_IDENT(a.attname) AS col_name
    ,CASE WHEN STRPOS(UPPER(format_type(a.atttypid, a.atttypmod)),
↪'CHARACTER VARYING') > 0
     THEN REPLACE(UPPER(format_type(a.atttypid, a.atttypmod)),
↪'CHARACTER VARYING', 'VARCHAR')
     WHEN STRPOS(UPPER(format_type(a.atttypid, a.atttypmod)),
↪'CHARACTER') > 0
     THEN REPLACE(UPPER(format_type(a.atttypid, a.atttypmod)),
↪'CHARACTER', 'CHAR')
     ELSE UPPER(format_type(a.atttypid, a.atttypmod))
     END AS col_datatype
    ,CASE WHEN format_encoding((a.attencodingtype)::integer) = 'none'
     THEN 'ENCODE RAW'
     ELSE 'ENCODE ' + format_encoding((a.attencodingtype)::integer)
     END AS col_encoding
    ,CASE WHEN a.atthasdef IS TRUE THEN 'DEFAULT ' + adef.adsrc ELSE
↪'' END AS col_default
    ,CASE WHEN a.attnotnull IS TRUE THEN 'NOT NULL' ELSE '' END AS␣
↪col_nullable
```

(continues on next page)

(continued from previous page)

```
    FROM pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    INNER JOIN pg_attribute AS a ON c.oid = a.attrelid
    LEFT OUTER JOIN pg_attrdef AS adef ON a.attrelid = adef.adrelid␣
↪AND a.attnum = adef.adnum
    WHERE c.relkind = 'r'
     AND a.attnum > 0
    ORDER BY a.attnum
    )
    --CONSTRAINT LIST
    UNION (SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,200000000 + CAST(con.oid AS INT) AS seq
    ,'\t,' + pg_get_constraintdef(con.oid) AS ddl
    FROM pg_constraint AS con
    INNER JOIN pg_class AS c ON c.relnamespace = con.connamespace␣
↪AND c.oid = con.conrelid
    INNER JOIN pg_namespace AS n ON n.oid = c.relnamespace
    WHERE c.relkind = 'r' AND pg_get_constraintdef(con.oid) NOT LIKE
↪'FOREIGN KEY%'
    ORDER BY seq)
    --CLOSE PARENT COLUMN LIST
    UNION SELECT c.oid::bigint as table_id,n.nspname AS schemaname,␣
↪c.relname AS tablename, 299999999 AS seq, ')' AS ddl
    FROM pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    WHERE c.relkind = 'r'

    --DISTSTYLE
    UNION SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,300000001 AS seq
    ,CASE WHEN c.reldiststyle = 0 THEN 'DISTSTYLE EVEN'
    WHEN c.reldiststyle = 1 THEN 'DISTSTYLE KEY'
    WHEN c.reldiststyle = 8 THEN 'DISTSTYLE ALL'
    WHEN c.reldiststyle = 9 THEN 'DISTSTYLE AUTO'
    ELSE '<<Error - UNKNOWN DISTSTYLE>>'
    END AS ddl
    FROM pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    WHERE c.relkind = 'r'
    --DISTKEY COLUMNS
    UNION SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,400000000 + a.attnum AS seq
    ,' DISTKEY (' + QUOTE_IDENT(a.attname) + ')' AS ddl
    FROM pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    INNER JOIN pg_attribute AS a ON c.oid = a.attrelid
    WHERE c.relkind = 'r'
    AND a.attisdistkey IS TRUE
```

(continues on next page)

(continued from previous page)

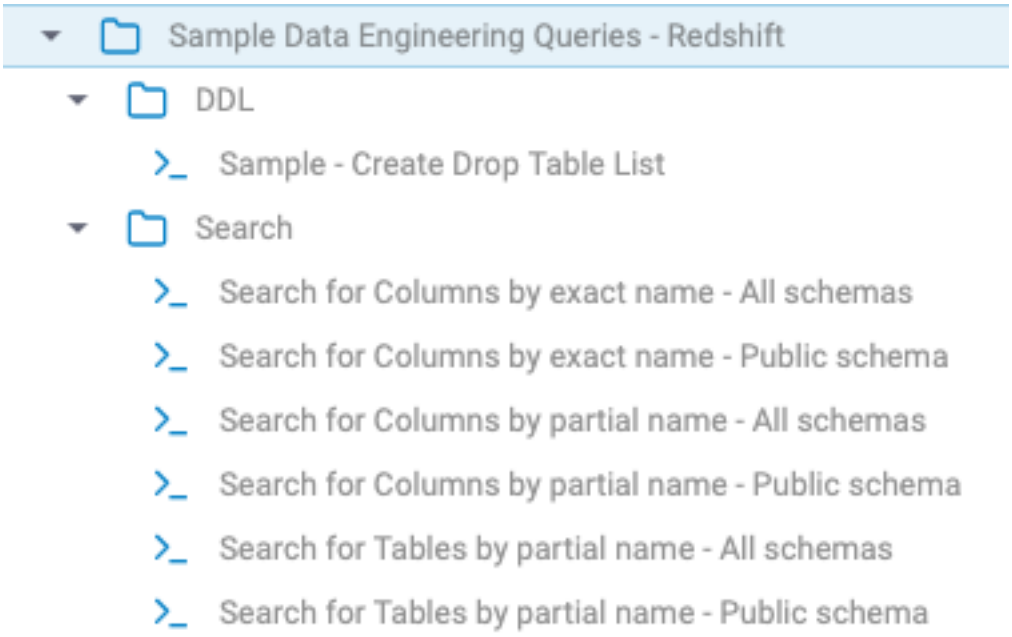
```
    AND a.attnum > 0
    --SORTKEY COLUMNS
    UNION select table_id,schemaname, tablename, seq,
       case when min_sort <0 then 'INTERLEAVED SORTKEY (' else '
↪SORTKEY (' end as ddl
    from (SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,499999999 AS seq
    ,min(attsortkeyord) min_sort FROM pg_namespace AS n
    INNER JOIN  pg_class AS c ON n.oid = c.relnamespace
    INNER JOIN pg_attribute AS a ON c.oid = a.attrelid
    WHERE c.relkind = 'r'
    AND abs(a.attsortkeyord) > 0
    AND a.attnum > 0
    group by 1,2,3,4 )
    UNION (SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,500000000 + abs(a.attsortkeyord) AS seq
    ,CASE WHEN abs(a.attsortkeyord) = 1
    THEN '\t' + QUOTE_IDENT(a.attname)
    ELSE '\t, ' + QUOTE_IDENT(a.attname)
    END AS ddl
    FROM  pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    INNER JOIN pg_attribute AS a ON c.oid = a.attrelid
    WHERE c.relkind = 'r'
    AND abs(a.attsortkeyord) > 0
    AND a.attnum > 0
    ORDER BY abs(a.attsortkeyord))
    UNION SELECT
    c.oid::bigint as table_id
    ,n.nspname AS schemaname
    ,c.relname AS tablename
    ,599999999 AS seq
    ,'\t)' AS ddl
    FROM pg_namespace AS n
    INNER JOIN  pg_class AS c ON n.oid = c.relnamespace
    INNER JOIN  pg_attribute AS a ON c.oid = a.attrelid
    WHERE c.relkind = 'r'
    AND abs(a.attsortkeyord) > 0
    AND a.attnum > 0
    --END SEMICOLON
    UNION SELECT c.oid::bigint as table_id ,n.nspname AS schemaname,
↪c.relname AS tablename, 600000000 AS seq, ';' AS ddl
    FROM  pg_namespace AS n
    INNER JOIN pg_class AS c ON n.oid = c.relnamespace
    WHERE c.relkind = 'r'

    )
    ORDER BY table_id,schemaname, tablename, seq
    )
    where schemaname = $schema and tablename = $table_name
```

Note: You will be prompted to supply an exact schema and table_name in this example.

2.1.4 Redshift Utilities Aginity Catalog Assets

There are seven assets you can add to your catalog. I chose to add them as shown below.



These queries are specific to Redshift but could be patterned after for other Database platforms. Just select a Redshift connection in the Pro/Team Editor and either double click the catalog item and execute or drag and drop the catalog item which will expose the code and run them.

Search for Columns by exact name - All schemas

This asset will search the information schema for columns with an exact name matches across all schemas

```
1  select table_catalog as database, table_schema, table_name, column_name,␣
   ↪ordinal_position as column_sequence
2  from information_schema.columns
3  where lower(column_name) = $column_name
```

Search for Columns by exact name - Public schema

This asset will search the information schema for columns with an exact name matches across just the public schema

```
1  select table_catalog as database, table_schema, table_name, column_name,␣
   ↪ordinal_position as column_sequence
2  from information_schema.columns
3  where lower(table_schema) = 'public'
4  and lower(column_name) = $column_name
```

Search for Columns by partial name - All schemas

This asset will search the information schema for columns with a partial name matches across all schemas

```
1  select table_catalog as database, table_schema, table_name, column_name,␣
   ↪ordinal_position as column_sequence
2  from information_schema.columns
3  where lower(table_schema) = 'public'
4  and lower(column_name) = $column_name
```

Search for Columns by partial name - Public schema

This asset will search the information schema for columns with an partial name matches across just the public schema

```
1  select table_catalog as database, table_schema, table_name, column_name,␣
   ↪ordinal_position as column_sequence
2  from information_schema.columns
3  where lower(table_schema) = 'public'
4  and lower(column_name) like $partial_column_name
```

Search for Tables by partial name - All schemas

This asset will search the information schema for tables with a partial name matches across all schemas

```
1  select table_catalog as database, table_schema, table_name
2  from information_schema.tables
3  where lower(table_name) like lower($partial_table_name)
```

Search for Tables by partial name - Public schema

This asset will search the information schema for tables with an partial name matches across just the public schema

```
1  select table_catalog as database, table_schema, table_name
2  from information_schema.tables
3  where lower(table_schema) = 'public'
4  and lower(table_name) like $partial_table_name
```

2.2 Lesson #3: Generating SQL to Profile Table Data

If you are here to use the data-profiling-SQL-generating example provided with Aginity Pro, you can jump straight to *Step 4 - Aginity Active Catalog Example*. If you want to dig into the concepts behind that example, then read on.

A large part of the day-to-day work of both data analysts and data engineers is to explore data in database tables. For small datasets, analysts have many tools we can use. In addition to showing us 5 number summaries, they might display distribution graphs or plots of relationships between columns. However, in the world of cloud-based data these tools might not be available. The data might be too big to bring back to the tool. We might have other restrictions related to connections or even legal concerns with transferring data. Sometimes, in order to begin exploring our data, we need to begin that exploration directly in SQL.

Of course, SQL is great for aggregating data. MPP cloud databases like Redshift, Snowflake, Netezza, and Hive are optimized for responding to this kind of SQL query. SQL is the most straight-forward language for expressing ideas

around data relationships and performing manipulations on data. However, there aren't generally-available facilities to explore "all" the data. For example, In Python, R, or SAS, there are language keywords that allow me to say "apply this function to *all* of the columns of this dataset. On the other hand,in SQL we have to specifically list each column transformation that we want to be performed.

All database systems have the concept of "system tables" or "system catalog" or "catalog tables" that are tables or views that contain information about all of the database objects, including table and column names. In this tutorial, we'll learn how to use these system tables to automatically generate SQL that can summarize our data.

2.2.1 Step 1 - What do you mean "generate SQL"?

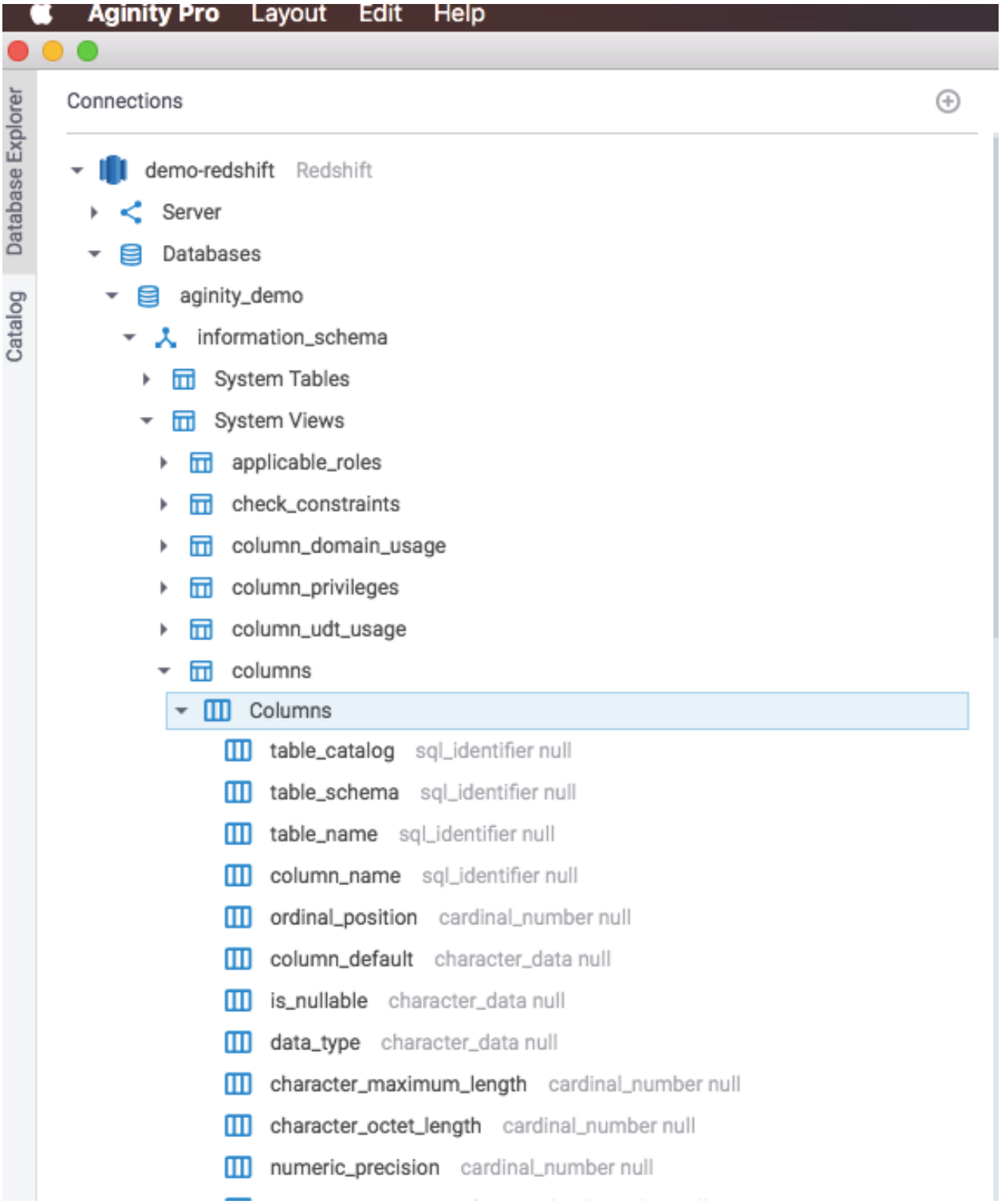
I love programming languages. A lot of languages are great for generating source code, either of their own language or of a different language. SQL is not a programming language that naturally comes to mind when we think about doing a lot of string manipulation. However, database administrators have a long history of writing SQL that generates SQL.

You might run into the term "dynamic SQL". This is a technical term for database systems that include a procedural language. That procedural language is used to generate SQL that will be submitted later within the same procedural language program.

For our purposes, we are technically using "static SQL", that is, when the SQL statement is submitted to the database system, it is a complete SQL statement. However, before submitting that SQL statement, we will submit a prior statement that will generate as the statement output, the static SQL that will profile our data. This idea will become clearer as we work through some examples.

2.2.2 Step 2 - What are "system tables"?

Just one more term before we jump into an example. Database "system tables" or "system views" hold information about the database itself. In Redshift (and in most database systems based on PostgreSQL), the view `information_schema.columns` contains all of the columns for all of the tables in the currently active database.



As an initial example, let's say that we want to generate a SQL statement to get the maximum value from each column in the `stl_plan_info` table that has "node" in its name. Here is a query to get the column names we are interested in:

```
select column_name, data_type
from information_schema.columns
```

(continues on next page)

(continued from previous page)

```
where table_name = 'stl_plan_info'
and data_type = 'integer'
and column_name like '%node%'
;
```

Output | Result 1

column_name	data_type
plannode	integer
nodeid	integer

Using some SQL string concatenation, we can generate the aggregation SQL based on those criteria:

```
select
'select '||quote_literal(column_name)||' as col_name, max('||column_name||') as max_
↪value from stl_plan_info;' as generated_sql
from information_schema.columns
where table_name = 'stl_plan_info'
and data_type = 'integer'
and column_name like '%node%'
;
```

Output | Result 1

generated_sql
select 'plannode' as col_name, max(plannode) as max_value from stl_plan_info;
select 'nodeid' as col_name, max(nodeid) as max_value from stl_plan_info;

You'll notice that this naive example produces one SQL statement per column, which might not be what we want. We'll revisit this issue later.

Note: A note on the examples used here.

Redshift includes STL Tables that contain log information for events that occur on the system. Some of these tables are only accessible to administrators. However, STL Tables pertaining to the queries that **you** execute are available to **you**. In order for you to follow along with this tutorial, when we need to profile some data we'll point to one of these STL Tables that is guaranteed to be available.

Method

As you can guess from looking at the brief example above, SQL that generates SQL can get complicated pretty quickly. The general method for developing these queries is first to figure out what you want the profiling SQL to look like.

Then, write a simple query to generate just one portion of that target SQL. Keep adding to your query until you achieve the target SQL.

In this case, we have a wonderfully complex SQL-generating query provided by Aginity as an example in the Active Catalog. Below, we'll look at some general principles, then explore this example query.

2.2.3 Step 3 - Data Profiling

If we already know which columns of a table are "interesting", then we can just write SQL to explore those columns. However, when we are presented with a new table, we don't know which columns are interesting and which are not. We don't know whether a column contains just a few values repeated over and over, or whether there are millions of unique values. We don't know whether the date columns represent only today, or whether they stretch back 20 or 30 years. Getting this knowledge over the data is one reason that we profile.

Another reason that we profile tables is to get a handle on data quality. If one day, our `STATE` table has 50 unique values and the next day, it has 482 unique values, then we might need to investigate the ETL process because something has clearly gone wrong. Other changes are more subtle. If the average `transaction_count` is 4,927,642 one day and it is 3,477,923 the next, then is there a problem with the data? Maybe, maybe not. However, we can capture the profile data each date to store in a table. Then we can check the standard deviation for the `average_transaction_count` to see whether there might be a problem worth investigating.

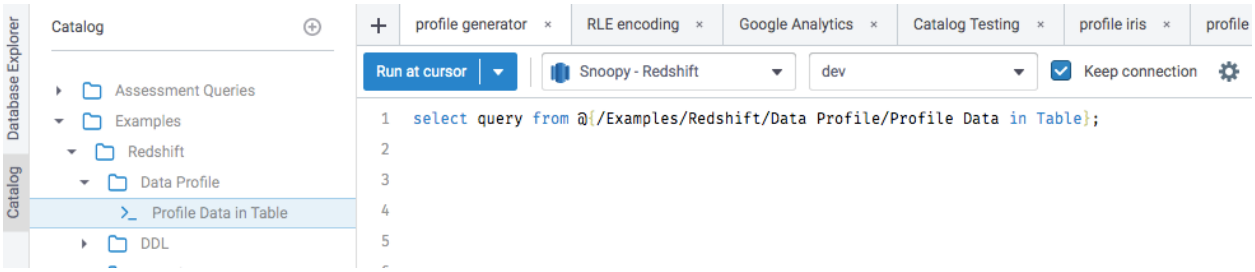
Data Types

This might seem obvious, but it bears emphasizing. Different data types have different aggregation functions. Taking the `average` (mean) of a list of dates doesn't make any sense. You might be able to get the `max` and `min` of a character string, but that doesn't really give the same insight as the same aggregation over numeric data.

A final consideration is that physical data types are not always useful for determining aggregation. For example, the *ID* column of a table might be of type `bigint`. You can take the min, max, and average of this column, but that doesn't tell you anything useful about the data. So, feel free to create your own version of this example – that's why Aginity provided it – that takes into account your local naming standards and business knowledge to avoid performing aggregation on columns that won't provide useful information.

2.2.4 Step 4 - Aginity Active Catalog Example

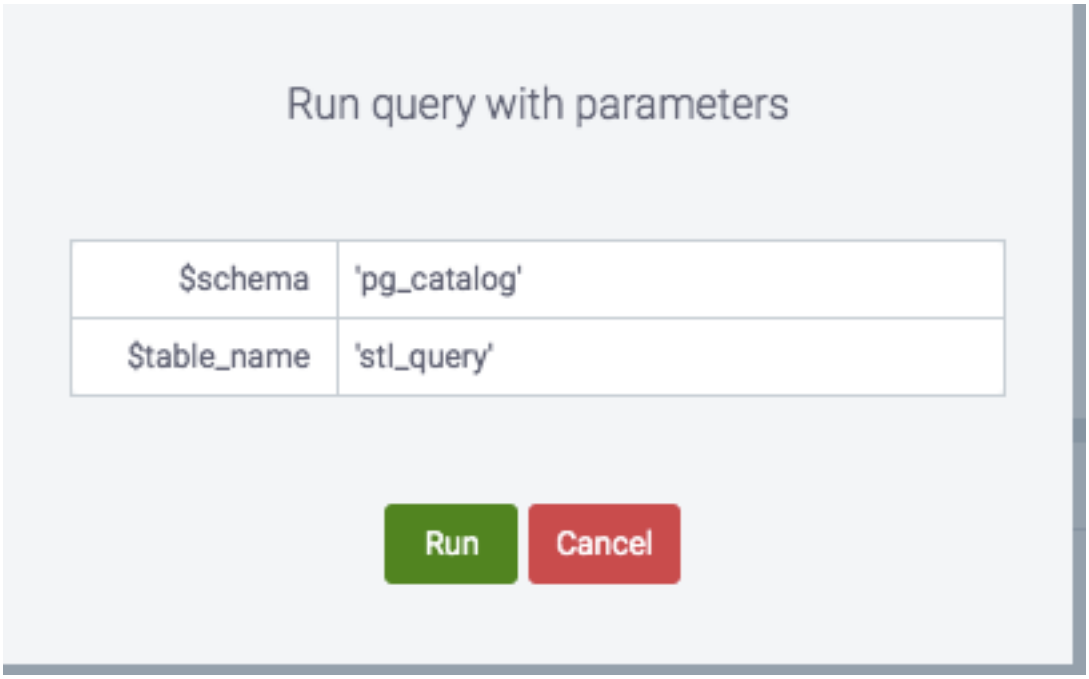
In Aginity Pro, in the "Catalog" tab, you should see a folder labeled `Sample Catalog`. Open the folder path `Sample Catalog -> Examples -> Redshift -> Data Profile`.



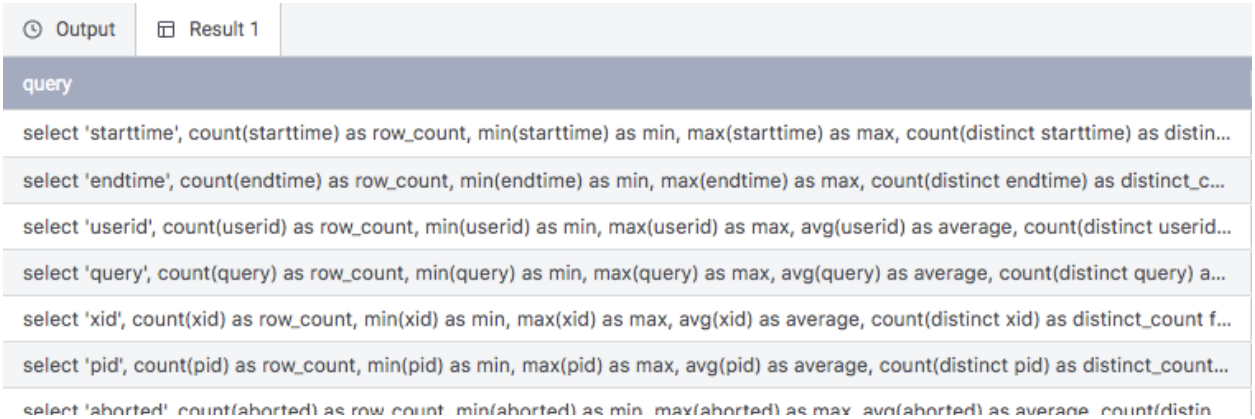
Set up your Query Window with this query:

```
select query from @{/Sample Catalog/Examples/Redshift/Data Profile/Profile Data in␣
↪Table};
```

and run with these parameters:



Your results will look like this:



with the "query" column containing the text of the data profiling SQL.

Here is the SQL that was returned with some formatting applied.

```
SELECT 'starttime',
       count(starttime) AS ROW_COUNT,
       min(starttime) AS MIN,
       max(starttime) AS MAX,
       count(DISTINCT starttime) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'endtime',
       count(endtime) AS ROW_COUNT,
       min(endtime) AS MIN,
       max(endtime) AS MAX,
       count(DISTINCT endtime) AS distinct_count
FROM pg_catalog.stl_query;

```

(continues on next page)

(continued from previous page)

```
SELECT 'userid',
       count(userid) AS ROW_COUNT,
       min(userid) AS MIN,
       max(userid) AS MAX,
       avg(userid) AS average,
       count(DISTINCT userid) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'query',
       count(query) AS ROW_COUNT,
       min(query) AS MIN,
       max(query) AS MAX,
       avg(query) AS average,
       count(DISTINCT query) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'xid',
       count(xid) AS ROW_COUNT,
       min(xid) AS MIN,
       max(xid) AS MAX,
       avg(xid) AS average,
       count(DISTINCT xid) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'pid',
       count(pid) AS ROW_COUNT,
       min(pid) AS MIN,
       max(pid) AS MAX,
       avg(pid) AS average,
       count(DISTINCT pid) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'aborted',
       count(aborted) AS ROW_COUNT,
       min(aborted) AS MIN,
       max(aborted) AS MAX,
       avg(aborted) AS average,
       count(DISTINCT aborted) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'insert_pristine',
       count(insert_pristine) AS ROW_COUNT,
       min(insert_pristine) AS MIN,
       max(insert_pristine) AS MAX,
       avg(insert_pristine) AS average,
       count(DISTINCT insert_pristine) AS distinct_count
FROM pg_catalog.stl_query
UNION ALL
SELECT 'concurrency_scaling_status',
       count(concurrency_scaling_status) AS ROW_COUNT,
       min(concurrency_scaling_status) AS MIN,
       max(concurrency_scaling_status) AS MAX,
       avg(concurrency_scaling_status) AS average,
       count(DISTINCT concurrency_scaling_status) AS distinct_count
FROM pg_catalog.stl_query;


```

(continues on next page)

(continued from previous page)

```
SELECT 'label',
       count(label) AS ROW_COUNT,
       max(top10_literals) AS top10_literals,
       count(DISTINCT label) AS distinct_count
FROM pg_catalog.stl_query r
CROSS JOIN
  (SELECT listagg(label, ',') top10_literals
   FROM
     (SELECT top 10 label
      FROM
        (SELECT label,
                count(*) cnt
         FROM pg_catalog.stl_query
         GROUP BY label)
      ORDER BY cnt DESC)) AS rr
UNION ALL
SELECT 'database',
       count(DATABASE) AS ROW_COUNT,
       max(top10_literals) AS top10_literals,
       count(DISTINCT DATABASE) AS distinct_count
FROM pg_catalog.stl_query r
CROSS JOIN
(SELECT listagg(DATABASE, ',') top10_literals
   FROM
     (SELECT top 10 DATABASE
      FROM
        (SELECT DATABASE,
                count(*) cnt
         FROM pg_catalog.stl_query
         GROUP BY DATABASE)
      ORDER BY cnt DESC)) AS rr
UNION ALL
SELECT 'querytxt',
       count(querytxt) AS ROW_COUNT,
       max(top10_literals) AS top10_literals,
       count(DISTINCT querytxt) AS distinct_count
FROM pg_catalog.stl_query r
CROSS JOIN
  (SELECT listagg(querytxt, ',') top10_literals
   FROM
     (SELECT top 10 querytxt
      FROM
        (SELECT querytxt,
                count(*) cnt
         FROM pg_catalog.stl_query
         GROUP BY querytxt)
      ORDER BY cnt DESC)) AS rr ;
```

Here we see three SQL statements: one for the two *time* columns, one for the seven *numeric* columns, and one for the three *text* columns.

2.2.5 Step 5 - Digging into the Example

Let's open up the example. In the Active Catalog, navigate as before but rather than double clicking on *Profile Data in Table*, this time drag it into the Query Window. This will expand the Catalog Item so that it looks like this:

```
  (select
  case

      when section = 'numeric' then
          'select '''||column_name||''', count('||column_name||') as row_count, min('||column_name||') as min, max('||column_name||') as max, avg('||column_name||') as average,
              count(distinct '||column_name||') as distinct_count
          from '||$schema||'.'||$table_name
      when section = 'text' then
          'select '''||column_name||''', count('||column_name||') as row_count, max(top10_literals) as top10_literals,
              count(distinct '||column_name||') as distinct_count
          from '||$schema||'.'||$table_name||' r
          cross join (select listagg( '||column_name||', '','') top10_literals from (select top 10 '||column_name||' from (select '||column_name||', count(*) cnt from '||$schema||'.'||$table_name||' group by '||column_name||') order by cnt desc)) as rr '
      when section = 'datetime' then
          'select '''||column_name||''', count('||column_name||') as row_count, min('||column_name||') as min, max('||column_name||') as max,
              count(distinct '||column_name||') as distinct_count
          from '||$schema||'.'||$table_name
  end ||
  case when ordinal_position = (
  select max(ordinal_position) from
  (
      select column_name, ordinal_position, 'numeric'::varchar(50) section
      from information_schema.columns
      where table_schema = $schema
          and table_name = $table_name
          and data_type in ('bigint', 'double precision', 'integer', 'numeric', 'real', 'smallint')
              union all
      select column_name, ordinal_position, 'text'::varchar(50) section
      from information_schema.columns
      where table_schema = $schema
          and table_name = $table_name
          and data_type in ('"char"', 'character', 'character varying', 'text')
              union all
      select column_name, ordinal_position, 'datetime'::varchar(50) section
      from information_schema.columns
      where table_schema = $schema
          and table_name = $table_name
          and data_type in ('abstime', 'date', 'timestamp with time zone', 'timestamp without time zone')
  ) c2 where c2.section = c1.section ) then ';' else
  ' union all'  end as query, section, ordinal_position
  from
  (
      select column_name, ordinal_position, 'numeric'::varchar(50) section
      from information_schema.columns
      where table_schema = $schema
          and table_name = $table_name
          and data_type in ('bigint', 'double precision', 'integer', 'numeric', 'real', 'smallint')
              union all
```

(continues on next page)

(continued from previous page)

```
48      select column_name, ordinal_position, 'text'::varchar(50) section
49      from information_schema.columns
50      where table_schema = $schema
51          and table_name = $table_name
52          and data_type in ('"char"', 'character', 'character varying', 'text')
53              union all
54      select column_name, ordinal_position, 'datetime'::varchar(50) section
55      from information_schema.columns
56      where table_schema = $schema
57          and table_name = $table_name
58          and data_type in ('abstime', 'date', 'timestamp with time zone', 'timestamp␣
    ↪without time zone')
59    ) c1
60    ) r order by section , ordinal_position;
```

You can learn a lot by digging into this example and adapting it for your own purposes, creating your own Active Catalog entries. Here, I'll draw your attention to two particular aspects.

The problem of stopping

In our initial example, we generated one statement per "node" column. We might try to combine these into a single statement with this (broken) code:

```
select
'select '||quote_literal(column_name)||' as col_name, max('||column_name||') as max_
↪value from stl_plan_info union all' as generated_sql
from information_schema.columns
where table_name = 'stl_plan_info'
and data_type = 'integer'
and column_name like '%node%'
;
```

which, after formatting, produces this SQL

```
SELECT 'plannode'    AS col_name,
       max(plannode) AS max_value
FROM   stl_plan_info
UNION ALL
SELECT 'nodeid'    AS col_name,
       max(nodeid) AS max_value
FROM   stl_plan_info
UNION ALL
```

The final "UNION ALL" doesn't belong there. This is a common problem when generating "delimited" items: *you need to handle the last item slightly differently than the rest of the items.* How does our Active Catalog example handle this?

Look particularly at this construction, with the concatenation operator on line 38:

```
case .. end || case .. end as query
```

The first `case` statement sets up generation of the aggregate functions, very similarly to our simplified example, with one clause for each data type. The second `case` statement begins by selecting only the row with the `max(ordinal_position)`. `ordinal_position` is given to us by the `information_schema.columns` view as the ordering of column names in the table. So, this second clause finds the *last* column name and appends a semicolon (`;`) in that case; otherwise, it appends `' union all'`.

In our simplified example, which doesn't need the first `case statement` it looks like this:

```
select
'select '||quote_literal(column_name)||' as col_name,
    max('||column_name||') as max_value from stl_plan_info' ||
case when ordinal_position =
  (select max(ordinal_position) from
    (select column_name, ordinal_position from information_schema.columns
     where table_name = 'stl_plan_info' and data_type = 'integer' and column_name␣
↪like '%node%')
   )
  then ';' else ' union all'
  end as generated_sql
from information_schema.columns
where table_name = 'stl_plan_info'
and data_type = 'integer'
and column_name like '%node%'
order by ordinal_position
;
```

which produces the desired output giving us a single, well-constructed SQL statement covering both columns. In our simplified version, we only have one possible row in the subselect with `max(ordinal_position)`. Because the Active Catalog version is handling multiple data types, it can get multiple result rows. It uses a technique that we won't cover here called a correlated subquery to manage that situation, which requires the usage of the `c1` and `c2` aliases that appear on lines 59 and 80.

Rolling up using list_agg()

A final consideration with the Active Catalog version is the situation with the "top 10" most frequently occurring values for text data. Here is the generated code for the `querytxt` column:

```
select 'querytxt', count(querytxt) as row_count,  max(top10_literals) as top10_
↪literals,
            count(distinct querytxt) as distinct_count
        from pg_catalog.stl_query r
        cross join (select listagg( querytxt, ',') top10_literals
                    from
                     (select top 10 querytxt from
                          (select querytxt, count(*) cnt
                           from pg_catalog.stl_query group by querytxt
                          ) order by cnt desc
                     )
                   ) as rr ;
```

Let's examine this from the inside out. The innermost subselect returns the `count(*)` for our target text column. The next layer up uses a nice Redshift function TOP to get the top 10 querytxt values by count. But we want those querytxt values rolled up to a single row rather than on multiple rows. This is exactly what the LISTAGG function does. It takes the text rows from the querytxt column and concatenates them into a single long string that is named "top10_literals". This probably isn't too useful for a complex text value like querytxt, but LISTAGG is great to have in your bag of tricks.

2.2.6 Conclusion

First, **use the Active Catalog**. Seriously, right now open a Redshift connection tab in Aginity Pro, navigate to the `Profile Data in Table` object, double-click it, put *select* * in front of the inserted text, hit F5, and start using

the generated code.

The idea of SQL that generates SQL is kind of mind-bending, but it is a technique that has been in use for several decades. You can find examples of it all over the web for every conceivable database system. I always learn something from deconstructing other people's code.

When writing your own SQL-generating SQL, start slowly. Once you have a technique that works, put that into the Active Catalog so that you can find it when you need it and even share it with your teammates.

2.3 Lesson #4: How to hide a picture of a cat in Redshift

I didn't start out trying to put a picture of a cat into Redshift. My motivation isn't less strange though. The product manager of Aginity Team put out a call for "interesting" things to do with Aginity Team. I thought, "what could be more interesting than having Aginity Team print a picture of Aginity's Enterprise Product Manager and Director of Solution Architecture, George L'Heureux?" Of course, encouraging you to put a picture of George, or of any of your co-workers into your Redshift database, would just be creepy. So, you can use an ASCII picture of a cat, hence the title of this article. The desktop software, Aginity Pro, works identically to Aginity Team, so you should be able to follow this tutorial with either version of software.

The truth is that those of us who wrangle data are often asked to do seemingly strange things with the tools at hand: query the Google Analytics API from Redshift; pull data from a GIS system directly into Snowflake; build an ad hoc ETL system inside Hive while we are waiting of IT to start the "real" project; etc. In this article, we will take some arbitrary string data, compress it according to a well-known algorithm by using some advanced SQL windowing functions, store the results in a database, read the compressed data, and uncompress the data using Reshift's regular expression syntax. The fact that the string data makes a picture of a cat, or of your co-worker, is just for fun. Along the way, we'll overcome some particular challenges that Redshift has in looping over functions.

2.3.1 Step 1 - Getting your string input

I don't have any particular advice on an ASCII art converter program. Search Google and you will find many of them. I kept my resulting output to 100 columns wide and 51 rows, which worked well for my image. Also I stayed away from output that included single quotes. Here is the image I was working with.

```
/////:::::::::::::::::-------------------------------------:::::::::::::::::::::////
↪////////++++++
/:::::::::::::-----------------------------------------------
↪::::::::::::::::::::://///////++++
::::::::::::-----------------------------://+ossshho+//--------------
↪::::::::::::::::::::::::://///////
::::::::-----------------------------:/ohdNNNMNMMMMMMNNNdy+:--------------
↪::::::::::::::::::::::://///
::::-----------------------.....--:sdNNNMMMMMMMMMMMMMMMMMMNd+------------------
↪:::::::::::::::::::////
---------------------..........-sdNMMMMNMMMMMMNNNMMMMNNMMMMMMMNd+----------------------
↪:::::::::::::::://
----------------.............-odMMMMMNNNmNmmdddmmmmddmmNNNNNMNdo:---------------------
↪--::::::::::::
-------------.............-sdNMMNNmmdhhhhssoosyyssssssyhhmmNNMNms:-------------------
↪----::::::::::
----------.................:yNMMNNmdhyssooo++++++++++++ossyyhmNMMNd+------------------
↪-------:::::::
------.....................-hNMMMmhyysoo++++//////////++++oosyhdmNMMNy--..------------
↪---------:::::
-----.....................+NMMNmhysso+++/////////:::////++++osyhdmNNNNs-.....---------
↪-----------:::
```

(continues on next page)

(continued from previous page)

```
---.....................sNMMNdysso+++/////////////////++++oooyhdNMMMm/.......-------
 ↪-----------::
-.....................-mMMMNdyso++++/////////////////++++osyhmNMMNs-.........----
 ↪--------------
-.....................:NMMMNdyso+++///::::-::/::::-:::///++osyhmNNMNh:...........--
 ↪--------------
.....................-mMMNNmysoo++//::----.-:------::////++syhmNMMNh:............
 ↪--------------
.....................-dNMNNmhysoooo++++++/:-:::///+oooo+osssshmNNNms-............
 ↪..------------
.....................+NNNNmhyssyyo++///://::::///://///+shyydNNNm+..............
 ↪...-----------
.....................-mNNNmhssys+++sys/+//////+o+++/yhyssoosyhNNNh:..............
 ↪....----------
.....................+NNNmysooosysss+///+o++++so/////++ooo+oymNms...............
 ↪.......-------
.....................:hmNms++++++//////++o+++oso+/::///+++oosdNmo...............
 ↪........------
.....................:oyNms++++////////++oo+/oso+/::::///++oodNd+...............
 ↪........------
......................./ymdooo+//::::::/+oo+/:/oso+:-----:/+syhmh/...............
 ↪........------
.....................smdsso+/:---://++++/:-:+ooo+/:---::/oyhdy:...............
 ↪.........----
.....................-ydyso//::::://:+o+++++ssss+///////oyhdo-...............
 ↪.........----
.....................-sdhyo++//+oossyyysso++syyyyyssso++oshmms-...............
 ↪..........---
.....................omdsoo+/+yyhhhhysso+/+oyyhdhddhs+ssydmmh/...............
 ↪.........----
.....................-ommhsss/+shyhhy+/:::::/:/osysyhsyshdmmmd+...............
 ↪.........-.--
.....................+mmdhyyyoos+/oys+/-...:/+ysoyysyhhddmmmd+-..............
 ↪..........---
.....................-ymmdhhhhyy+//+sssoooossyoooshyhddmmNmmy/-..............
 ↪..........---
......................./mNmdhhhhyo/:/+++++++oo++oyyhhddmmNNmd+-..............
 ↪........-..--
.....................-hNNmdddds/:/+o++++++++o+shdmmmNNNmmy:.................
 ↪.........-..-
.....................:ymmNNmmmmdyo//+osssssso+/ohdmNNNNNNmy/.................
 ↪.........----
.....................-+ymNhomNNNNmmdhsoosssoo+syssydmNNNNNNmh+.................
 ↪.........----
.....................-:/oydmNNNNy-+dmNNNmmmdhhyyysooyhhddmNNNNNNNNmdyo:-.............
 ↪..........---
.................-:/oydmNNNNNNNNms..-+dmNNNmmmmddddhdmmmmmNNNmhNNNNNNNmdhs+:-.........
 ↪...........--
.........--/+oyhdmNNNNNNNNNNNNNms..``-sdmmNNNNmNmmmNNNNmNNNmy/+mmNNNNNNNNNNmdyo+/:-...
 ↪..........---
--..-:/oyhdmNNNNNNNNNNNNNNNNNNmms..````-ohdmmNNNNNNNNmNNmy+:..smmNNNNNNNNNNNNNNNmmdyo/
 ↪:-.........---
-/+shdNNNNNNNNNNNNNNNNNNNNNNNNNNmms..```  `.:osyhhddddhyy+:.```-
 ↪hmmmmmNNNNNNNNNNNNNNNNNNmdyo/:--------
dmNNNNNNNNNNNNNNNNNNmNNNNNNNNNNNmmy.`````    `.:/+//:-.````````-
 ↪dmmmmmmmmmmmNNNNNNNNNNNNNNNNNNNmhy+:----
NNNNNNNNNNNNNNNNNNmNNNNNNNNNNNmmmh.`````````  `.-::``   `````  `/
 ↪mmmmmmmmmmmNNmNNNNNNNNNNNNNNNNNNNmhs/-
```

(continues on next page)

(continued from previous page)

```
NNNNNNNNNNNNNNNNNNNmNNNNNNNNNmmmd.` `````````shddy:   `````
 ↪`ommmmmmmmmmmmmNNNNNmNNNNNNNNNNNNNNNNNNNms
NNNNNNNNNNNNNNNNNNNNNNNNNNNmmmmmmm-``    ```.ymmhyh:````
 ↪`ymmmNmmmmmmmmNNNNNNNmNNNNNNNNNNNNNNNNNNm
NNNNNNNNNNNNNNNNNNNNNNNNNmNNmmmmmmm:``    ``.`:Ndyhd/``        `.
 ↪mmmmNmmmmmmmNNNNNNNNmNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNmmmmmmmmmm+`````````+Ndhdh``        ``/
 ↪mmmmmmNmmmNNmNNNNNNmmNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNmNNNNNNmmNmmmmms`````````ymhddh-``   `
 ↪````smmmmmmNmmNNNNNNNNNNNmNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNmNNmmmmmh.``````.omdhmdhd:````````.
 ↪dmmNmmmNmmNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNmNNmNmmd.``````/hmhhmmymh.
 ↪```````:mmmNmmNNmNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNmNNNmNmm:`````.
 ↪smdhhdmhmd:```````smmmmmNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNmmNNmmNm+`````:dNdhhdddmd/```.``.
 ↪hmmNmNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNmNNNmmms.```.+mmhsyyddmd+`..
 ↪```:mmNmmNNmNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNmh..`..hmmhhyhmmmds`..
 ↪```ommNmNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

You can use whatever techniques you like for getting data into Redshift. The COPY command is very handy. Here is what I did:

```
drop table my_text;
create table my_text (seq int IDENTITY, line varchar(255));

copy my_text
from 's3://your_s3_bucket/txt/my_picture_ascii.txt'
credentials 'aws_access_key_id=XXXXXXXXX;aws_secret_access_key=ZZzzZZZzzZZZZ'
format as FIXEDWIDTH 'line:100'
;
```

However, note the sentences in the CREATE TABLE documentation that say:

```
With a COPY operation, the data is loaded in parallel and distributed to the node␣
 ↪slices.
To be sure that the identity values are unique, Amazon Redshift skips a number of␣
 ↪values
when creating the identity values.  As a result, identity values are unique and␣
 ↪sequential,
but not consecutive, and the order might not match the order in the source files.
```

What I found is that with a small file like this, although there are gaps in the `seq` numbers, as long as I `order by seq`, my lines are correct. If you run into problems, then you can add a sequence number to your text file to keep things straight.

2.3.2 Step 2 - Encoding the Data

We don't want to just insert our text picture into a table and then pull it back out. Where's the challenge in that? We need to encode it somehow. I looked at the DEFLATE algorithm, but decided not to tackle encoding that in SQL today. Our data is plain ASCII with a lot of repeats. After a bit of searching, Run Length Encoding seemed like a good choice for this exercise.

Run Length Encoding

For Run Length Encoding, we are going to take a string, such as the first line of our ASCII picture:

```
/////:::::::::::::::::----------------------------------------:::::::::::::::::::::////
↪////////++++++
```

and encode all of the repeats. I found several implementations online and it seems pretty loose as to whether the character or number is first. For our purposes, we'll put the character first. So the encoding for this line is:

```
/5:16-39:22/12+6
```

because the string has 5 slashes followed by 16 semicolons, followed by 39 dashes, etc.

Iterating in Redshift

The Redshift **substring** function will be useful here,

```
select seq, line, substring(line, 1, 1) as letter
from my_text
order by seq
;
```

Output | Result 1

seq	line	letter
103	/////::::::::::::::::----------	/
105	/::::::::::::::----------	/
107	::::::::::::----------	:
109	::::::::----------	:
111	::::----------	:
113	----------	-

That gives us the first letter of each line. What we really want is each letter of each line turned into rows of our query result. For example, Hive has functions that can pull a string apart into an array and can "explode" that array into rows of a query result. Redshift doesn't have that, so we will have to handle that ourselves.

In order to explode our letters from the string into rows, we need to iterate over each letter. We want a row for each letter of the original string. We can use the SQL **cross join** against a "numbers" table to achieve this effect.

Note: Sidebar *numbers*

Needing a set of numbers is so common in SQL that many database systems have a function to generate numbers as needed. For example, PostgreSQL 9 has *generate_series()*. Unfortunately, Redshift and many other MPPs lack this feature; fortunately in MPPs, we have lots of things we can count in order to generate numbers.

In Redshift, the **stl_plan_info** table has a record for every query you have run. If you have been using Redshift for a while, then this willl be sufficient. Our string is only 100 characters wide, so we are only going to use the numbers

from 1 through 100.

```
drop table if exists numbers;
create temp table numbers as
select num from
(select cast(row_number() over (partition by 1) as int) as num from stl_plan_info
) inner_query
;
```

To achieve this iteration, we can modify our query like this:

```
select seq, num, line, substring(line, num, 1) as letter
from my_text
cross join numbers
where num <= length(line)
order by seq, num
;
```

Output | Result 1

seq	num	line	letter
103	1	/////::::::::::::::::----------...	/
103	2	/////::::::::::::::::----------...	/
103	3	/////::::::::::::::::----------...	/
103	4	/////::::::::::::::::----------...	/
103	5	/////::::::::::::::::----------...	/
103	6	/////::::::::::::::::----------...	:
103	7	/////::::::::::::::::----------...	:
103	8	/////::::::::::::::::----------...	:

For each row in the *my_text* table, the **cross join** will give us a copy for each row of our *numbers* table. Our **substring** now includes **num** from the *numbers* table. We don't need copies for *all* of the rows in the numbers table, so we limit it to only the length of the text. Now we have our string pivoted so that each letter is on a separate row.

Where are we heading? We want to count the 5 slashes followed by the 16 dashes and so on down the result set.

Grouping with windowing functions

Redshift supports the standard SQL Window functions. My first thought was to use **count()** or **row_number()** grouped over our letters. However, letters that we have already seen can return later in a line. For example, the slash characters that begin the first line also end the first line. This return of characters foils any attempt to use *only* grouping functions. First, we need to mark each change, preserving the character number where the change occurs. We use the LAG() function to bring in the previous letter for comparison and then some **case** logic to determine that a change has happened.

```
select seq, num, line, substring(line, num, 1) as letter,
   lag(letter) over(partition by seq order by num) as previous_letter,
   case when num = 1 then 1
        when letter <> previous_letter then num
   else 0 end as group_start
from my_text
```

(continues on next page)

(continued from previous page)

```
cross join numbers
where num <= length(line)
order by seq, num
;
```

Output | Result 1

seq	num	line	letter	previous_letter	group_start
2	1	/////::::::::::::::::----------	/		1
2	2	/////::::::::::::::::----------	/	/	0
2	3	/////::::::::::::::::----------	/	/	0
2	4	/////::::::::::::::::----------	/	/	0
2	5	/////::::::::::::::::----------	/	/	0
2	6	/////::::::::::::::::----------	:	/	6
2	7	/////::::::::::::::::----------	:	:	0

We want to use the group_start along with the next_group_start using the LEAD() function, filtering out all of the rows that don't start a new group. Now we have one row for each group. Each of those rows has sufficient information to calculate the beginning and end of a substring for that group, which we'll call a "chunk".

```
select seq, group_start,
       nvl(lead(group_start) over(partition by seq order by group_start),␣
↪length(line)+1) as next_group_start,
       substring(line, group_start, next_group_start - group_start) as chunk,
       substring(chunk, 1, 1) || length(chunk) as encoded_chunk
from
(select seq, num, line, substring(line, num, 1) as letter,
       lag(letter) over(partition by seq order by num) as previous_letter,
       case when num = 1 then 1
            when letter <> previous_letter then num
       else 0 end as group_start
    from my_text
    cross join numbers
    where num <= length(line)
    order by seq, num
 )
where group_start <> 0
order by seq, group_start
;
```

Output | Result 1

seq	group_start	next_group_start	chunk	encoded_chunk
2	1	6	/////	/5
2	6	22	::::::::::::::::	:16
2	22	61	----------...-	-39
2	61	83	::::::::::::::::::::	:22
2	83	95	///////////	/12
2	95	101	++++++	+6
4	1	2	/	/1

Rolling up with the listagg function

Now we have each chunk "encoded" with the character and count. We only need to bring all of the encoded chunks up to the same row so that the encodings for the entire line are together. The Redshift LISTAGG() function is what we want here.

```
select seq as id, listagg(encoded_chunk) within group (order by group_start) as value
from (
    select seq, group_start,
           nvl(lead(group_start) over(partition by seq order by group_start),␣
↪length(line)+1) as next_group_start,
           substring(line, group_start, next_group_start - group_start) as chunk,
           substring(chunk, 1, 1) || length(chunk) as encoded_chunk
    from
    (select seq, num, line, substring(line, num, 1) as letter,
           lag(letter) over(partition by seq order by num) as previous_letter,
           case when num = 1 then 1
                when letter <> previous_letter then num
           else 0 end as group_start
        from my_text
        cross join numbers
        where num <= length(line)
        order by seq, num
     )
    where group_start <> 0
    order by seq, group_start
)
group by seq
order by seq
;
```

Output | Result 1

id	value
2	/5:16-39:22/12+6
4	/1:13-52:20/10+4
6	:12-30:1/2+1o1s3h2o1+1/2-1...
8	:8-29:1/1o1h1d1N3M1N1M6N...
10	:4-23.5-2:1s1d1N3M18N1d1+...
12	-21.11-1s1d1N1M4N1M5N3M...
14	-17.13-1o1d1M5N3m1N1m2d...
16	-14.14-1s1d1N1M2N2m2d1h4...
18	-10.17:1y1N1M2N2m1d1h1y1s...

We have now encoded our string lines into an RLE format. Before we decode, let's store that in a table:

```
create table dev.public.rle_data as
select * from ...
;
```

2.3.3 Step 3 - Decoding the Data

Now our picture is "hidden" in a Redshift table, with each line compressed into RLE format. We used iteration over the **substring()** function plus some windowing functions to get there. To pull the data out, we are going to use iteration over Redshift regular expression functions.

Here is our first row of data:

```
/5:16-39:22/12+6
```

We have a target character, followed by a number that tells how many times to repeat the character. We do have a REPEAT() function in Redshift.

```
select repeat('/', 5);
```

So, if we can identify the components, we can build the string for that repetition. We know from the previous section that if we can get those repetition strings into columns, we can use the LISTAGG() function to pull them together into the same row, reconstituting the original string.

Iterating using Regular Expression functions

Redshift supports several regular expression functions. Understanding regular expressions is one of the most important skills a programmer can have. With modern database systems, that includes SQL programmers. MPP SQL systems vary widely in their support for regular expressions. In Redshift, the REGEXP_SUBSTR() function is straight-forward but limited.

```
select *, REGEXP_SUBSTR(value, '\\D\\d{1,}', 1, 1)
from rle_data a
order by id
;
```

Output | Result 1

id	value	regexp_substr
1	/5:16-39:22/12+6	/5

With this function we are substringing the value column, which contains the RLE encoded string, using the pattern `\\D\\d{1,}`, starting from the beginning of the string, and extracting the first occurrence of the string, which is `/5`. Developing regular expression patterns begins with thinking clearly about what you are trying to achieve. In this case, we set up our RLE to be "a character followed by some digits". Another way to characterize that statement is "a non-digit followed by at least 1 digit", which is represented by the pattern `\\D\\d{1,}`.

With Hive, we could "explode" directly into an array with multiple occurrences of the regular expression. With Redshift, we'll use the previous technique of using a numbers table to drive iteration over the *occurrence* parameter of REGEXP_SUBSTR() . The only thing we need to know is when to stop iterating, which could be different for each string. Fortunately, the REGEXP_COUNT() function will help with that.

```
select *, REGEXP_SUBSTR(value, '\\D\\d{1,}', 1, num)
from rle_data a
cross join numbers n
where num <=REGEXP_COUNT(value, '\\D\\d{1,}')
order by id, num
;
```

Output | Result 1

id	value	num	regexp_substr
1	/5:16-39:22/12+6	1	/5
1	/5:16-39:22/12+6	2	:16
1	/5:16-39:22/12+6	3	-39
1	/5:16-39:22/12+6	4	:22
1	/5:16-39:22/12+6	5	/12
1	/5:16-39:22/12+6	6	+6
2	/1:13-52:20/10+4	1	/1
2	/1:13-52:20/10+4	2	:13

For each pattern occurrence, we want to expand it using the `REPEAT()` function shown above. For that function, we need the first character of the occurrence and the number of times to repeat, which is everything in the pattern occurrence **after** the first character. Let's look at all of that data on a single row:

```
select row_number() over (partition by 1) as key,
       id as line_number, num as pat_number, value as rle_line,
       REGEXP_SUBSTR(value, '\\D\\d{1,}', 1, num)  as pattern_occurrence,
       substring(pattern_occurrence, 1, 1) as rle_char,
       cast(substring(pattern_occurrence, 2) as int) as char_count,
       repeat(rle_char, char_count) as expanded_pattern
from
 (select *, REGEXP_SUBSTR(value, '\\D\\d{1,}', 1, num)
  from rle_data a
  cross join numbers n
  where num <=REGEXP_COUNT(value, '\\D\\d{1,}')
  order by id, num
 ) as rle_data
order by id, num
 ;
```

Output | Result 1

key	line_number	pat_number	rle_line	pattern_occurrence	rle_char	char_count	expanded_pattern
1	1	1	/5:16-39:22/12+6	/5	/	5	/////
66	1	2	/5:16-39:22/12+6	:16	:	16	::::::::::::::::
52	1	3	/5:16-39:22/12+6	-39	-	39	----------
117	1	4	/5:16-39:22/12+6	:22	:	22	::::::::::::::::::::
173	1	5	/5:16-39:22/12+6	/12	/	12	//////////
239	1	6	/5:16-39:22/12+6	+6	+	6	++++++
27	2	1	/1:13-52:20/10+4	/1	/	1	/
92	2	2	/1:13-52:20/10+4	:13	:	13	:::::::::::::

Rolling up

Once again, we can use the `LISTAGG()` function to roll this data up to a single row.

```
select line_number, rle_line, listagg(expanded_pattern) within group (order by line_
↪number, pat_number) as full_line
from
    (select row_number() over (partition by 1) as key,
           id as line_number, num as pat_number, value as rle_line,
           REGEXP_SUBSTR(value, '\\D\\d{1,}', 1, num)  as pattern_occurrence,
           substring(pattern_occurrence, 1, 1) as rle_char,
           cast(substring(pattern_occurrence, 2) as int) as char_count,
           repeat(rle_char, char_count) as expanded_pattern
    from
     (select *, REGEXP_SUBSTR(value, '\\D\\d{1,}', 1, num)
      from rle_data a
      cross join numbers n
      where num <=REGEXP_COUNT(value, '\\D\\d{1,}')
      order by id, num
     ) as rle_data
    order by id, num
    ) rle_detail
group by line_number, rle_line
order by line_number
;
```

Output | Result 1

line_number	rle_line	full_line
1	/5:16-39:22/12+6	/////::::::::::::::::----------...
2	/1:13-52:20/10+4	/::::::::::::::----------...
3	:12-30:1/2+1o1s3h2o1+1/2-1...	::::::::::::----------...
4	:8-29:1/1o1h1d1N3M1N1M6N...	::::::::----------...
5	:4-23.5-2:1s1d1N3M18N1d1+...	::::----------.....
6	-21.11-1s1d1N1M4N1M5N3M...	----------.....-...
7	-17.13-1o1d1M5N3m1N1m2d...	----------.....-od...
8	-14.14-1s1d1N1M2N2m2d1h4...	----------.....-sdN...

Now the `full_line` column has your complete picture, decoded from RLE format.

2.3.4 Conclusion

You will probably never need to hide an ASCII picture of a cat (or a co-worker) inside your Redshift database. However, you will certainly need to iterate over subsets of data, use windowing functions to group data, and use regular expressions to manipulate strings.

As modern data analysts and data engineers, we need to expand our toolbox to include all types of string manipulations. The next time you are asked to do something out of the ordinary in Redshift, Snowflake, or any other cloud database, you will be ready.