SedonaOffice Users Conference

San Francisco, CA | January 21 – 24, 2018

Data Mining Part 2

Presented by:
Matt Howe

PERENNIAL SOFTWARE

This Page Intentionally Left Blank

Table of Contents

Overview

In this session, we are going to continue from part 1, adding commands and options to write more advanced queries to extract data from a SedonaOffice Database. We will still be using SQL or Structured Query Language. If you have not looked at part 1 or at least have some knowledge of SQL, this course material may not be for you. We will be starting where we left off in part 1.

Note: All SQL keywords, queries and code snippets will be in *italics* to make them stand out from the commentary.

Part 1 Example Continued

In part 1, we developed a query to see how many appointments our technician, Ralph completed in a given month.

```
select
c.Customer_Number,
b.Business_Name,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description,
csy.Alarm_Account,
sy.Description as 'System Type',
std.Arrival_Time,
std.Departure_Time,
r.Description
from AR_Customer c
inner join AR_Customer_Bill b on c.Customer_Id = b.Customer_Id and b.Is_Primary =
'Y'
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
inner join AR_Customer_System csy on si.Customer_Site_Id = csy.Customer_Site_Id
inner join SY_System sy on csy.System_Id = sy.System_Id
inner join SV_Service_Ticket st on csy.Customer_System_Id = st.Customer_System_Id
inner join SV_Service_Ticket_Dispatch std on st.Service_Ticket_Id =
std.Service_Ticket_Id
inner join SV_Service_Tech stt on std.Service_Tech_Id = stt.Service_Tech_Id
inner join SV_Resolution r on std.Resolution_Id = r.Resolution_Id
inner join SY_Employee e on stt.Employee_Id = e.Employee_Id
Where std.Arrival_Time > {d'2016-06-01'} and std.Arrival_Time < {d'2016-07-01'}
and e.Employee_Code = 'Ralph'
```

Notes

Functions

Functions accept parameters, operate on them and return a value. Let us start by looking at some time functions.

DateDiff	Returns amount of time elapsed between two date times
DateAdd	Adds an amount of time to a date time
DatePart	Returns a single number representing a part of a date time

All of these refer to a specific part of the date time. IE. Year, days, minutes, etc. Here is a list of the more common parts.

Year	yy or yyyy
Month	m, mm
Day	d, dd
Hour	h, hh
Minute	m, mm

We want to see how long Ralph was on these calls. So, we will use the DateDiff function. For the parameters, it wants to know the units we want, the start date and the end date. For this, we want minutes (minute in this case, not one of the abbreviations), the arrival time and the departure time. We will include the full date times because the function is smart enough to handle a span that crosses midnight.

```
select
c.Customer_Number,
b.Business_Name,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description,
csy.Alarm_Account,
sy.Description as 'System Type',
std.Arrival_Time,
std.Departure_Time,
DATEDIFF(minute, std.Arrival_Time,std.Departure_Time) as 'Minutes on Site',
r.Description
from AR_Customer c
inner join AR_Customer_Bill b on c.Customer_Id = b.Customer_Id and b.Is_Primary =
'Y'
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
inner join AR_Customer_System csy on si.Customer_Site_Id = csy.Customer_Site_Id
inner join SY_System sy on csy.System_Id = sy.System_Id
inner join SV_Service_Ticket st on csy.Customer_System_Id = st.Customer_System_Id
inner join SV_Service_Ticket_Dispatch std on st.Service_Ticket_Id =
std.Service_Ticket_Id
inner join SV_Service_Tech stt on std.Service_Tech_Id = stt.Service_Tech_Id
inner join SV_Resolution r on std.Resolution_Id = r.Resolution_Id
inner join SY_Employee e on stt.Employee_Id = e.Employee_Id
Where std.Arrival_Time between {d'2016-06-01'} and {d'2016-07-01'}
and e.Employee_Code = 'Ralph'
```

This returns a column, which we have named “Minutes on Site”, showing us how many minutes Ralph was there. However, what if we want the time in hours and minutes? Well we could create a column for hours and one for minutes like this.

```
DATEDIFF(minute, std.Arrival_Time,std.Departure_Time)/60 as 'Hours on Site',
DATEDIFF(minute, std.Arrival_Time,std.Departure_Time)%60 as 'Minutes on Site',
```

Because the numbers returned are integers, we can divide minutes by 60 to get hours and use the % operator to get the reminder in minutes. However, what if we want one column that looks like 1:43 for one hour and 43 minutes? We can use a cast function. The cast function takes values of one type and displays them as another type. In our case, we want to take integer numbers and display them as text. This is a double line. To SQL it is one line but I wrote it as multiple lines to make it more easily read.

```
cast(DATEDIFF(minute, std.Arrival_Time,std.Departure_Time)/60 as nvarchar(2)) + ':' +
right('0' + cast(DATEDIFF(minute, std.Arrival_Time,std.Departure_Time)%60 as
nvarchar(2)),2) as 'Time on site',
```

The first line calculates the hours on site as we saw before, but then returns it to us as a two-character string. We add (concatenate) that string to a string holding our colon ':' and then add it to the next line. In line two, we add a leading zero to the remaining minutes cast as a string. So 9 becomes 09 and 45 becomes 045, We then use the *Right* to get only the right two characters so 09 is 09 and 045 is 45.Add it all together and we get.

```
select
c.Customer_Number,
b.Business_Name,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description,
csy.Alarm_Account,
sy.Description as 'System Type',
std.Arrival_Time,
std.Departure_Time,
cast(DATEDIFF(minute, std.Arrival_Time,std.Departure_Time)/60 as nvarchar(2)) + ':' +
right('0' + cast(DATEDIFF(minute, std.Arrival_Time,std.Departure_Time)%60 as
nvarchar(2)),2) as 'Time on site',
r.Description
from AR_Customer c
inner join AR_Customer_Bill b on c.Customer_Id = b.Customer_Id and b.Is_Primary =
'Y'
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
inner join AR_Customer_System csy on si.Customer_Site_Id = csy.Customer_Site_Id
inner join SY_System sy on csy.System_Id = sy.System_Id
inner join SV_Service_Ticket st on csy.Customer_System_Id = st.Customer_System_Id
inner join SV_Service_Ticket_Dispatch std on st.Service_Ticket_Id =
std.Service_Ticket_Id
inner join SV_Service_Tech stt on std.Service_Tech_Id = stt.Service_Tech_Id
inner join SV_Resolution r on std.Resolution_Id = r.Resolution_Id
inner join SY_Employee e on stt.Employee_Id = e.Employee_Id
Where std.Arrival_Time between {d'2016-06-01'} and {d'2016-07-01'}
and e.Employee_Code = 'Ralph'
```

Notes

Example 2

In this example, we are going to count the number of service tickets each customer site had in the 2016. But only in certain zip codes and only if we actually went on the call. In addition, we want all of the customer sites in that area even if their service ticket total was zero.

So, we will design a query to gather the site information.

```
select
c.Customer_Number,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description
from AR_Customer c
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
```

Now we need to add a where clause. We are looking for all of the sites in zip codes 07001 through 07009. We could make a complex series of Or clauses, but instead let us use *like*. *Like* uses wild cards to match patterns; the wild card operators are listed below.

Wildcard	**Description**	**Example**
%	Any string of zero or more characters.	WHERE title LIKE '%computer%' finds all book titles containing the word 'computer'.
_ (Underscore)	Any single character.	WHERE au_fname LIKE '_ean' finds all four-letter first names that end with ean (Dean, Sean, etc.)
[ ]	Any single character within the specified range ([a-f]) or set ([abcdef]).	WHERE au_lname LIKE '[C-P]arsen' finds author last names ending with arsen and starting with any single character between C and P, for example Carsen, Larsen, Karsen, and so on. In range searches, the characters included in the range may vary depending on the sorting rules of the collation.
[^]	Any single character not within the specified range ([^a-f]) or set ([^abcdef]).	

Examining our list we see each zip code starts with the same first characters so our check could be ‘0700%’ or ‘0700_’. I personally like to use % whenever possible as I think it is easier to read.

One last option is to use the *In* keyword. That checks against a list of values. So our options are a huge *Or* block, the *Like* comparison or the *In* comparison. But we have chosen the *Like* method.

```
select
c.Customer_Number,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description
from AR_Customer c
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
Where GE3_Description like '0700%'
```

Notes

Sub-Queries and Aggregates

Now we need to create a query to return the Service_Ticket_Id of every dispatch in 2016.

```
Select std.Service_Ticket_Id from SV_Service_Ticket_Dispatch std where
std.Arrival_Time between {d'2016-01-01'} and {d'2017-01-01'}
```

Now we can use the list returned to us for an *In* filter.

```
select
count(st.Service_Ticket_Id)
from SV_Service_Ticket st
Where st.Service_Ticket_Id in
(Select std.Service_Ticket_Id from SV_Service_Ticket_Dispatch std where
std.Arrival_Time between {d'2016-01-01'} and {d'2017-01-01'})
```

Remember the *In* comparison requires a list, but it does not have to be a list you typed in. So here we a query that contains another query. That contained query is called a sub- query and we are going to use it again in just a couple of minutes. Also, this shows us an aggregate function *Count*. This is the first aggregate we have used so let us stop and talk a bit about them. Aggregates examine a number of records and then return a single bit of data about them. Below is a list of aggregates and what they do.

Count()	Counts the field inside the ()
Sum()	Totals the numeric values in the field inside the ()
Avg()	Returns the average value of the numeric field inside the ()
Max()	Returns the maximum value of the field inside the ()
Min()	Returns the minimum value of the field inside the ()
Var()	Returns the value of the difference between Max and Min

So we have a query that lists the sites we want, and a query that counts service tickets we actually went out on, now let us combine them.

Notes

```
select
c.Customer_Number,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description,
(select
count(st.Service_Ticket_Id)
from SV_Service_Ticket st
Where st.Service_Ticket_Id in
(Select std.Service_Ticket_Id from SV_Service_Ticket_Dispatch std where
std.Arrival_Time between {d'2016-01-01'} and {d'2017-01-01'}
and st.Customer_Site_Id = si.Customer_Site_Id)) as 'Service Ticket Count'
from AR_Customer c
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
Where GE3_Description like '0700%'
```

We can again place a query in a query as a sub-query, give it a title and it shows just like a regular column. We add this to the sub-query where clause to make sure we are counting the correct site's ticekets.

```
and st.Customer_Site_Id = si.Customer_Site_Id
```

Unions

Wouldn't be nice to have a totals row at the bottom? By now you should already be able to create a totals query so I'm not going to cover that in depth. Here is what I came up with for a totals query.

```
select
'Total Sites',
Cast((Select count(Customer_Site_Id) from AR_Customer_Site where GE3_Description
like '0700%') as nvarchar(5)),
'', '', '', 'Total Tickets',
(select
count(st.Service_Ticket_Id)
from SV_Service_Ticket st
Where st.Service_Ticket_Id in
(Select std.Service_Ticket_Id from SV_Service_Ticket_Dispatch std where
std.Arrival_Time between {d'2016-01-01'} and {d'2017-01-01'})
and st.Customer_Site_Id in (select Customer_Site_Id from AR_Customer_Site Where
GE3_Description like '0700%'))
```

Notice I placed a number of empty columns (‘’) in the query and cast the count sites as nvarchar. That is because I wanted the same number and type of columns in our totals query as we had in the detail query. This is so we can join the two queries together using the *Union* keyword like so.

```
select
c.Customer_Number,
si.Business_Name,
si.Address_1, si.GE1_Description, si.GE2_Description, si.GE3_Description,
(select
count(st.Service_Ticket_Id)
from SV_Service_Ticket st
Where st.Service_Ticket_Id in
(Select std.Service_Ticket_Id from SV_Service_Ticket_Dispatch std where
std.Arrival_Time between {d'2016-01-01'} and {d'2017-01-01'}
and st.Customer_Site_Id = si.Customer_Site_Id)) as 'Service Ticket Count'
from AR_Customer c
inner join AR_Customer_Site si on c.Customer_Id = si.Customer_Id
Where GE3_Description like '0700%'
Union
select
'Total Sites',
Cast((Select count(Customer_Site_Id) from AR_Customer_Site where GE3_Description
like '0700%') as nvarchar(5)),
'', '', '', 'Total Tickets',
(select
count(st.Service_Ticket_Id)
from SV_Service_Ticket st
Where st.Service_Ticket_Id in
(Select std.Service_Ticket_Id from SV_Service_Ticket_Dispatch std where
std.Arrival_Time between {d'2016-01-01'} and {d'2017-01-01'})
and st.Customer_Site_Id in (select Customer_Site_Id from AR_Customer_Site Where
GE3_Description like '0700%'))
```

The *Union* keyword joins two queries together into a single record set if they have the same number and type of columns. If we had a third query we could add that in also with another *Union*. This is very powerful. In the dashboard is a query that unions together 18 separate queries and is 379 lines long, but it was developed by this same method. Each of the small queries were developed separately and then “unioned” together. Anyone of you here is now capable of making that query now.

Virtual Tables

Sometimes you have a complex query with unions and sub-queries. This is great, has date ranges etc., however, what if you want months with different date ranges? You would need to go into every union and change the date range, very easy to make a mistake. Temp tables help with this. Let us make a query that covers how much a customer was invoiced, broken out by invoice type. Now we could use a number of sub-queries to find this, however every sub-query would have a date range and editing them would be difficult. So, we create a query to bring back everything and then use it as the table in another query. So, first step, create the invoice query.

```
Select
c.Customer_Id,
i.Type_JSCO,
i.Amount,
i.Invoice_Date
From AR_Customer c
left Outer join AR_Invoice i on c.Customer_ID = i.Customer_Id
union
Select
c.Customer_Id,
'Deposit',
d.Amount,
d.Transaction_Date
From AR_Customer c
left Outer join AR_Deposit_Check d on c.Customer_ID = d.Customer_Id
```

I am not going to explain this query, as there is nothing new.

Now let us use this query as a virtual table.

```
Select
cu.Customer_Number,
cu.Customer_Name,
inv.Type_JSCO,
sum(inv.Amount) As 'Amount'
from
(Select
c.Customer_Id,
i.Type_JSCO,
i.Amount,
i.Invoice_Date
From AR_Customer c
left Outer join AR_Invoice i on c.Customer_ID = i.Customer_Id
union
Select
c.Customer_Id,
'Deposit',
d.Amount,
d.Transaction_Date
From AR_Customer c
left Outer join AR_Deposit_Check d on c.Customer_ID = d.Customer_Id) inv
inner join AR_Customer cu on cu.Customer_Id = inv.Customer_Id
Where inv.Invoice_Date between {d'2016-06-01'} and {d'2016-06-30'}
Group By cu.Customer_Number, cu.Customer_Name, inv.Type_JSCO
Order by cu.Customer_Number, cu.Customer_Name, inv.Type_JSCO
```

First, we see our original query with union inside parenthesis, and named “inv“. We can now use it just like any normal table. We can even join it to other normal tables as we see directly below the virtual table. Above the virtual table we see a normal query selecting fields from our two tables (virtual and normal). However, there is even more new stuff in this query.

Notes

Group By, Having and Order By

Below the *Where* clause we see a new keyword *Group By*. This works with the aggregate sum in the Select portion. Before we always used aggregates in a sub-query where we summed the entire record set. But here we want to have the sum based on customer number and transaction type. So we tell SQL to sum based on these fields. When creating a *Group By*, every field that is not an aggregate needs to be in the *Group By*. Finally, if we want to make sure our query comes out in a particular order, we use the *Order By* keyword. Following the *Order By* is the list of fields we wish to order by and an optional *ASC* or *DESC* parameter controlling the type of order (ascending or descending). If no parameter is given, it defaults to ASC.
If we only wanted to see Job and Service type invoices, we can add the clause:

```
Having inv.Type_JSCO in ('J', 'H', 'S')
```

Because a job invoice can be a type “J” or “H” (hold back) we need to check for both along with “S” for service. I decided to use the *In* keyword check to filter the results. The value before the keyword *In* is checked against the list after the keyword *In*. The list must be in parenthesis and if it is text, each entry must be in single quotes.

Also notice I used *Having* instead of *Where*. Any filtering done before the Group By is a Where clause. Any filtering done after the Group By is Having. The difference is how SQL treats the filter. The Where clause reduces the number of records that will be operated on and thus making the query execute faster. But, the Having clause because it works after the grouping and aggregates, can filter records like Sum(Amount) > 1200. The Having I created here for demonstration purposes should really be in the Where clause.

Notes

Views and Select Into

We are briefly going to discuss two other options for making your query semi-permanent. The first method is to change it into a view. Views are fixed queries that act like a table. Again, you can join to them and use them in other queries.

The major differences between views and queries are, first, they are faster. SQL pre-compiles them and stores them ready to go. Regular queries are compiled every time you run them. Secondly, they are not easily changed. If we wanted to make our virtual table into a view, we would do this as shown below.

```
Create View Matt_Invoice_Deposit as

Select
c.Customer_Id,
i.Type_JSCO,
i.Amount,
i.Invoice_Date
From AR_Customer c
left Outer join AR_Invoice i on c.Customer_ID = i.Customer_Id
union
Select
c.Customer_Id,
'Deposit',
d.Amount,
d.Transaction_Date
From AR_Customer c
left Outer join AR_Deposit_Check d on c.Customer_ID = d.Customer_Id
```

The top line does everything for you. Notice I prefaced it with my name. You sould develop a standard preface for your queries to make sure you do not interfere with the built in queries. If you want to change the view, you have to drop it first.

```
Drop view Matt_Invoice_Deposit
```

This is why prefacing your queries is important. You would hate to break SedonaOffice by accidently dropping the wrong view.

You can also use your query to create a table. This code would create a query from our virtual table query.

```
Select inv.Customer_Id, inv.Type_JSCO, inv.Amount, inv.Invoice_Date
into Matt_Invoice_Deposit
from
(Select
c.Customer_Id,
i.Type_JSCO,
i.Amount,
i.Invoice_Date
From AR_Customer c
left Outer join AR_Invoice i on c.Customer_ID = i.Customer_Id
union
Select
c.Customer_Id,
'Deposit',
d.Amount,
d.Transaction_Date
From AR_Customer c
left Outer join AR_Deposit_Check d on c.Customer_ID = d.Customer_Id) inv
```

Notice the line *into Matt_Invoice_Deposit* this does the table building and contains the name of the table (Matt_Invoice_Deposit).

To get rid of the table when you are finished:

```
Drop table Matt_Invoice_Deposit
```

Notes

Additional Sample Queries

Finally, I am going to give you a few queries that people have found useful through the years. Feel free to use or modify them to fit your needs.

Get all customers whose annual RMR is between $239.00 and $245.00:

```
Select
c.Customer_Number,
b.Business_Name,
(Select Sum(r.Monthly_Amount*12) From AR_Customer_Recurring r where
r.Cycle_Start_Date <= GETDATE() And (r.Cycle_End_Date <= {d'1900-01-01'} Or
r.Cycle_End_Date > GETDATE())
And r.Customer_Id = c.Customer_Id) as Annual
From AR_Customer c
Inner Join AR_Customer_Bill b on c.Customer_Id = b.Customer_Id
Where b.Is_Primary = 'Y' And (Select Sum(r.Monthly_Amount*12) From
AR_Customer_Recurring r where
r.Cycle_Start_Date <= GETDATE() And (r.Cycle_End_Date <= {d'1900-01-01'} Or
r.Cycle_End_Date > GETDATE())
And r.Customer_Id = c.Customer_Id) Between 239.00 and 245.00
```

Get a range of service appointments and dispatch times:

```
SELECT t.Ticket_Number,
d.Schedule_Time,
d.Dispatch_Time,
e.Employee_Code,
c.Customer_Number,
c.Customer_Name
FROM SV_Service_Ticket t
INNER JOIN SV_Service_Ticket_Dispatch d ON t.Service_Ticket_Id =
d.Service_Ticket_Id
INNER JOIN SV_Service_Tech tech ON d.Service_Tech_Id = tech.Service_Tech_Id
INNER JOIN SY_Employee e ON tech.Employee_Id = e.Employee_Id
INNER JOIN AR_Customer c ON t.Customer_ID = c.Customer_ID
WHERE d.Schedule_Time >= {d'2013-01-01'} AND d.Schedule_Time < {d'2013-01-31'}
ORDER BY d.Schedule_Time
```

Get how much a customer paid last year in monitoring, service and installations:

```
Select
c.Customer_Number,
b.Business_Name,
(Select IsNull(SUM(i.Amount), 0) From AR_Invoice i
Where i.Type_JSCO = 'C' And i.Invoice_Date Between {d'2012-01-01'} And {d'2012-12-31'}
And i.Customer_Id = c.Customer_Id) as 'Monitoring',
(Select IsNull(SUM(i.Amount), 0) From AR_Invoice i
Where i.Type_JSCO = 'S' And i.Invoice_Date Between {d'2012-01-01'} And {d'2012-12-31'}
And i.Customer_Id = c.Customer_Id) as 'Service',
(Select IsNull(SUM(i.Amount), 0) From AR_Invoice i
Where i.Type_JSCO = 'J' And i.Invoice_Date Between {d'2012-01-01'} And {d'2012-12-31'}
And i.Customer_Id = c.Customer_Id) as 'Installs',
(Select IsNull(SUM(i.Amount), 0) From AR_Invoice i
Where i.Type_JSCO = 'O' And i.Invoice_Date Between {d'2012-01-01'} And {d'2012-12-31'}
And i.Customer_Id = c.Customer_Id) as 'Other'
From AR_Customer c
Inner Join AR_Customer_Bill b on b.Customer_Id = c.Customer_Id
```

Notes

Get a detailed list of cancelled accounts for sales to do follow up calls:

```
Select
cu.Customer_Number,
cu.Customer_Name,
st.Customer_Status_Code,
cb.Branch_Code as Customer_Branch,
ty.Type_Code,
cs.Business_Name,
cs.Address_1,
cs.Address_2,
cs.GE1_Description,
cs.GE2_Short,
cs.GE3_Description,
cs.Zip_Code_Plus4,
sb.Branch_Code as Site_Branch,
ts.System_Code,
pt.Panel_Type_Code,
cq.CS_Cancelled_Date,
cu.Customer_Since,
cq.Effective_Date,
cq.Reference,
cq.Memo,
it.Item_Code,
cr.Monthly_Amount,
cq.Balance_Of_Contract,
cq.Full_Cancel
From AR_Customer cu
Inner Join AR_Type_Of_Customer ty On cu.Customer_Type_Id = ty.Type_Id
Inner Join SS_Customer_Status st On cu.Customer_Status_Id = st.Customer_Status_Id
Inner Join AR_Customer_Site cs On cu.Customer_Id = cs.Customer_Id
Inner Join AR_Branch cb On cu.Branch_Id = cb.Branch_Id
Inner Join AR_Branch sb On cs.Branch_Id = sb.Branch_Id
Inner Join AR_Customer_System sy On cs.Customer_Site_Id = sy.Customer_Site_Id
Inner Join SY_System ts On sy.System_Id = ts.System_Id
Inner Join SY_Panel_Type pt On sy.Panel_Type_Id = pt.Panel_Type_Id
Inner Join AR_Customer_Recurring cr On sy.Customer_System_Id =
cr.Customer_System_Id
Inner Join AR_Item it on cr.Item_Id = it.Item_Id
Inner Join AR_Cancel_Queue cq On cu.Customer_Id = cq.Customer_Id
Inner Join AR_Cancel_Queue_Site qs On cq.Cancel_Queue_Id = qs.Cancel_Queue_Id
Where cr.Cycle_End_Date = cq.Effective_Date And cs.Customer_Site_Id =
qs.Customer_Site_Id
And cq.Effective_Date Between {d'2012-01-01'} And {d'2012-12-31'}
Order By cu.Customer_Number
```