Module 6: Modularizing Data Retrieval Logic and Creating a Star Schema

While not a requirement, it's usually easier and more efficient to modularize your reports by encapsulating your SELECT queries within *views*, *functions*, and/or *stored procedures*. Views, functions, and stored procedures are objects you create in the database and then access from reporting tools such as SQL Server Reporting Services.

In this module we'll first walk through the process of creating and using each of these types of objects. In addition, we'll look at some techniques for implementing these objects in scenarios where you do not have the necessary permissions to create objects in the database or schemas containing the data tables themselves.

Views

A view is, quite simply, a *virtual table*. It is used to simplify complex SELECT queries. Consider the following example:

```
CREATE VIEW dbo.vTotalMakeSales
AS
SELECT
       c.Make
       ,SUM(s.SalesAmount) as [TotalSales]
FROM dbo.Sales s
JOIN dbo.Cars c
ON s.CarID=c.CarID

GROUP BY c.Make;
```

This creates a view named **dbo.vTotalMakeSales**. We can browse a list of all user-defined views in the Views folder under the database folder in the Object Explorer panel of SSMS. While not a requirement, it's a best practice to choose a consistent naming convention for user-defined objects. We're going to preface all of our view names with a lower-case "v".

We query the view as if it were a table, as in the following example:

```
SELECT
       Make
       ,TotalSales
FROM dbo.vTotalMakeSales
```

However, the view is simply an abstraction of the SELECT query we used to define it. When we query the view, the query processor actually just runs the SELECT query we used to define the view, and treats the result set as if it were a table.

Also, there are a few restrictions on the SELECT query we use to define the view:

1. The view cannot contain an ORDER BY clause unless it also includes a TOP clause. This doesn't mean you can't use an ORDER BY clause when you query the view, just that the view itself can't contain one.
2. The INTO keyword or the OPTION clause (INTO is beyond the scope of this course).
3. A view cannot reference a temporary table or table variable (temporary tables and table variables are beyond the scope of this course).

A view can be modified using ALTER VIEW. In the following query we modify dbo.vTotalMakeSales by adding a WHERE clause:

```
ALTER VIEW dbo.vTotalMakeSales
AS
SELECT
	c.Make
	,SUM(s.SalesAmount) as [TotalSales]
FROM dbo.Sales s
JOIN dbo.Cars c
ON s.CarID=c.CarID
WHERE s.SalesDate >= '20120101' AND s.SalesDate < '20130101'
GROUP BY c.Make;
```

Finally, a view can be dropped (i.e. completely removed) as follows:

```
DROP VIEW dbo.vTotalMakeSales
```

Let's say we want to use the results of a complex query in a SQL Server Reporting Services report or perhaps make the results easily accessible to a Microsoft Excel user. In Module 5 we demonstrated a rather complicated SELECT query using two derived tables. If we want to make this query easier to access, we could simply define a view using it, as follows (notice we'll need to remove the original ORDER BY):

```
CREATE VIEW dbo.vCustomerAverages
AS
SELECT
	c.CustomerID
	,p.LastName
	,p.FirstName
	,cda.MaritalStatus
	,cda.RuralUrban
	,cda.MeanSalesAmount AS [Marital Rural/Urban Average]
	,ca.MeanSalesAmount AS [Customer's Average]
FROM Customers c
JOIN dbo.CustomerDemographics cd
ON c.DemographicsGroup=cd.DemogroupID
JOIN dbo.Person p
ON c.PersonID=p.PersonID
JOIN
(
SELECT
	c.CustomerID
	,AVG(s.SalesAmount) AS [MeanSalesAmount]
FROM dbo.Sales s
JOIN dbo.Customers c
```

```
ON s.CustomerID=c.CustomerID
GROUP BY c.CustomerID
) ca
ON c.CustomerID=ca.CustomerID
JOIN
(
SELECT
       cd.MaritalStatus
       ,cd.RuralUrban
       ,AVG(s.SalesAmount) AS [MeanSalesAmount]
FROM dbo.CustomerDemographics cd
JOIN dbo.Customers c
ON c.DemographicsGroup=cd.DemogroupID
JOIN dbo.Sales s
ON s.CustomerID=c.CustomerID
GROUP BY cd.MaritalStatus, cd.RuralUrban
) cda
ON cda.MaritalStatus=cd.MaritalStatus
AND cda.RuralUrban=cd.RuralUrban
```

When we want to use the result set we can query the view rather than run the underlying query, as follows:

```
SELECT * FROM dbo.vCustomerAverages
ORDER BY LastName, FirstName
```

We've added the original ORDER BY to the query against the view.

NOTE: Be careful when using the * wildcard rather than listing the column names. It's a best practice to retrieve only the data you actually need, and often the * wildcard will result in superfluous column data in your result set. It's OK in this context because the view was created to return specifically with the columns we want.

Table-Valued Functions

There are several different types of functions. We're going to limit our discussion to one specific type – the *inline table-valued* (ITV) function. The ITV function is similar to a view in that it allows us to abstract and modularize a SELECT query. Unlike a view, the ITV function can take one or more parameters. However, functions are not exposed to client tools such as Excel in the same way as tables (generally, client tools access tables and views in exactly the same way).

Like views, the functions you create become objects within the database. You can find a listing of all user-defined functions in the Programmability/Functions folder under the database in Object Explorer. The following query creates an inline table-valued function.

The following CREATE FUNCTION command will create a simple inline table valued function using a query we created in Module 2. We're going to preface all of our function names with *ufn_*.

```
CREATE FUNCTION dbo.ufn_GetNames
(
)
```

```
RETURNS TABLE
AS
RETURN
(
SELECT
	FirstName
	,LastName
	,LastName + ', ' + FirstName as [Full Name]
FROM dbo.Person
);
```

We can then call the function and retrieve our results set like this:

```
SELECT * FROM dbo.ufn_GetNames();
```

Note that even though this function does not take a parameter, we still need to include the empty parentheses after the function name in the SELECT query.

Consider the following query from Module 4:

```
SELECT
	c.Make
	,c.Model
	,c.Year
	,sum(s.SalesAmount) as [Total Sales]
FROM dbo.Sales s
JOIN dbo.Cars c
ON s.CarID=c.CarID
WHERE year(s.SalesDate) = 2012
GROUP BY c.Make, c.Model, c.Year
HAVING sum(s.SalesAmount) > 2000000
ORDER BY c.Make, c.Model, c.Year
```

As originally written, this query will always return only data from the year 2012. However, if we create an ITV function using this query we could pass in the year as a parameter, like this (notice we've removed the ORDER BY. Like views, ITV functions cannot contain ORDER BY without TOP):

```
CREATE FUNCTION dbo.ufn_GetSalesData
(
	@year int
)
RETURNS TABLE
AS
RETURN
(
SELECT
	c.Make
	,c.Model
	,c.Year
	,sum(s.SalesAmount) as [Total Sales]
FROM dbo.Sales s
JOIN dbo.Cars c
```

```
ON s.CarID=c.CarID
WHERE year(s.SalesDate) = @year
GROUP BY c.Make, c.Model, c.Year
HAVING sum(s.SalesAmount) > 2000000
)
```

Notice that to create a parameter we give the parameter a name that starts with @, and assign it a data type. We can then refer to the parameter anywhere inside the function, and the parameter will contain the passed-in value.

We would call the function and return a result set like this:

```
SELECT * FROM dbo.ufn_GetSalesData(2012)
```

The integer value 2012 will be passed in to the function and assigned to the parameter @year. @year can be used in place of any expected integer value anywhere within the function.

If we want be able to pass in a sales amount value for use in the HAVING clause, we can run an ALTER FUNCTION statement and add the additional parameter:

```
ALTER FUNCTION dbo.ufn_GetSalesData
(
	@year int,
	@lowestsalestotal money
)
RETURNS TABLE
AS
RETURN
(
SELECT
	c.Make
	,c.Model
	,c.Year
	,sum(s.SalesAmount) as [Total Sales]
FROM dbo.Sales s
JOIN dbo.Cars c
ON s.CarID=c.CarID
WHERE year(s.SalesDate) = @year
GROUP BY c.Make, c.Model, c.Year
HAVING sum(s.SalesAmount) > @lowestsalestotal
);
```

To drop a function we use DROP FUNCTION, as follows:

```
DROP FUNCTION dbo.ufn_GetSalesData
```

Stored Procedures

Stored procedures are another technique we might use to modularize and encapsulate our SELECT queries. They are particularly easy to use within SQL Server Reporting Services. Stored procedures are

used widely within database programming, as there are few limits to the SQL statements they can contain. We're going to limit our use of them to stored procedures that return single result sets. Like views and functions, the stored procedures we create become database objects. You can see a list of them in the Programmability\Stored Procedures folder under the database in Object Explorer.

Like functions, stored procedures can take parameters. Unlike functions, we execute stored procedures rather than SELECT from them. Here's a simple example using a SELECT query from Module 2:

```
CREATE PROCEDURE dbo.usp_GetNames
AS
SELECT
	PersonID
	,FirstName
	,MiddleInitial
	,LastName
	,RTRIM(LEFT(FirstName,1) + ISNULL(MiddleInitial,'')) + LEFT(LastName,1) as
[Initials]
FROM dbo.Person;
```

We call the stored procedure by executing it as follows:

```
EXEC dbo.usp_GetNames;
```

We can modify a stored procedure by using ALTER PROCEDURE as follows:

```
CREATE PROCEDURE dbo.usp_GetNames
AS
SELECT
	PersonID
	,FirstName
	,MiddleInitial
	,LastName
	,RTRIM(LEFT(FirstName,1) + ISNULL(MiddleInitial,'')) + LEFT(LastName,1) as
[Initials]
FROM dbo.Person;
```

We drop a stored procedure using DROP PROCEDURE as follows:

```
DROP PROCEDURE dbo.usp_GetNames;
```

The following creates a stored procedure using one of the SELECT queries from Module 4. The stored procedure takes two parameters, @year and @lowerlimit. These parameters work exactly like those in the function we created earlier.

Notice that there's no restriction on ORDER BY in the stored procedure.

```
CREATE PROCEDURE dbo.usp_GetSalesPersonReport
	@year int,
```

```
	@lowerlimit money
AS
SELECT
	p.LastName [Sales Person]
	,c.Make
	,sum(s.SalesAmount) as [Total Sales]
FROM dbo.Sales s
JOIN dbo.Cars c
on s.CarID=c.CarID
JOIN dbo.SalesPeople sp
on s.SalesPersonID=sp.EmployeeID
JOIN dbo.Person p
on sp.PersonID=p.PersonID
WHERE year(s.SalesDate) = @year
GROUP BY p.LastName, c.Make
HAVING sum(s.SalesAmount) > @lowerlimit
ORDER BY p.LastName, [Total Sales] DESC;
```

When we pass parameter values to the stored procedure we can do it in a couple of ways. First, we can explicity assign values to the parameters by name, as in the following:

```
EXEC dbo.usp_GetSalesPersonReport @year=2012, @lowerlimit=10000;
```

Alternatively, we can simply list the parameter values in order, as in the following:

```
EXEC dbo.usp_GetSalesPersonReport 2012,10000;
```

Creating a Star Schema

Many analytical platforms, such as Power BI, organize data into a logical structure known as a ***star schema***. The star schema is a way to represent data sets for efficient storage and analysis. In this section, we're going to create a star schema for our CarDealer database using views.

However, before we begin, let's create a **date table**. We're going to want to include this in our star schema, so let's go ahead and create it.

Creating a Date Table

A ***date table*** is a table in which each record represents an individual date, and each column represents an attribute of that date.

Below is an example of a date table. Note that each row represents an individual date, and each column represents an attribute of that date. For our purposes we'll limit the number of date attributes we represent, but date tables often include columns to indicate whether the day is a week day vs. weekend, a holiday, the day's location in the fiscal calendar, and other date-related information of importance to the organization.

	DateValue	YearofDate	QuarterofDate	MonthofYear	NameofMonth	DayOfMonth	NameofDay	DayofWeek	DayofYear	WeekofYear
1	1950-01-01	1950	1	1	January	1	Sunday	1	1	1
2	1950-01-02	1950	1	1	January	2	Monday	2	2	1
3	1950-01-03	1950	1	1	January	3	Tuesday	3	3	1
4	1950-01-04	1950	1	1	January	4	Wednesday	4	4	1
5	1950-01-05	1950	1	1	January	5	Thursday	5	5	1
6	1950-01-06	1950	1	1	January	6	Friday	6	6	1
7	1950-01-07	1950	1	1	January	7	Saturday	7	7	1
8	1950-01-08	1950	1	1	January	8	Sunday	1	8	2
9	1950-01-09	1950	1	1	January	9	Monday	2	9	2
10	1950-01-10	1950	1	1	January	10	Tuesday	3	10	2
11	1950-01-11	1950	1	1	January	11	Wednesday	4	11	2
12	1950-01-12	1950	1	1	January	12	Thursday	5	12	2
13	1950-01-13	1950	1	1	January	13	Friday	6	13	2
14	1950-01-14	1950	1	1	January	14	Saturday	7	14	2

There are multiple ways of creating a date table. There are many freely available and widely used SQL scripts you can find online and customize to your needs. In the lab for this module, for example, you're going to run an SQL script modified slightly from publicly-available source code. The date table will often contain far more rows of dates and columns of date attributes than will be needed for a given analysis, so much of the time we'll query our date table via a view.

Below, we're going to walk through the process of creating our own basic data table. While it's a bit simplistic compared to wide-available scripts, it does serve to illustrate the basics of writing your own SQL code to create a date table.

First, let's create it. Below is the CREATE TABLE statement we'll use:

```
create table dbo.Dates
(
	DateValue date,
	YearOfDate int,
	QuarterOfDate int,
	MonthOfYear int,
	NameOfMonth varchar(15),
	DayOfMonth int,
	NameOfDay varchar(15),
	DayOfWeek int,
	DayOfYear int,
	WeekOfYear int
)
```

The above statement will create a table named DimDate in the dbo schema, with columns and datatypes as listed in the statement.

A helpful practice is to check to see if a table already exists prior to running a CREATE TABLE statement. The following, for example, will check to see if a table named dbo.DimDate already exists, and if it does it will drop it.

This statement should of course precede CREATE TABLE.

```
drop table if exists dbo.Dates;
```

To add data to DimDate, we're going to use a **WHILE** loop. A WHILE loop will allow our TSQL to ***iterate***, or repeat, until a Boolean expression evaluates to false or null. It will have the following format:

```
WHILE (This expression is true)
BEGIN

        Keep running these TSQL commands

END
```

Before we implement the WHILE loop, we're going to create a couple of **variables** in which to store the start and the end of the range of dates we want to include in the table. Variable names must start with @, and a datatype must be assigned with the variable is declared. In this case, we've also assigned values to the variables.

Note about variables: Variables are scoped to the batch in which they are declared. This means any statements that use the variables must be run at the same time as the DECLARE statement that declared the variables.

```
DECLARE @StartDate date = '01/01/1950';
DECLARE @EndDate date = '01/01/2050';
```

Now we'll create a third variable to store the date value of the current iteration through the loop, and set it equal to the first day in the range:

```
DECLARE @currentDate date = @StartDate;
```

Next, we'll create a WHILE loop and run it once for each date in our range. We'll make the loop end after we've processed the last day of the year: With each iteration, we'll insert a row for the current value of @currentDate, then increment @currentDate by one day. When @currentDate reaches @EndDate, the WHILE loop ends.

```
WHILE @currentDate < @EndDate
BEGIN

        --This is where we need to put the INSERT statement.
        --It should insert a row for the date currently being processed.

        set @currentDate = DATEADD(day,1,@currentDate);

END
```

With each iteration of the loop, we'll insert a row into the dbo.Date table. We're going to use the year, DATEPART, and DATENAME functions to generate attributes of that date, such as the name of the day, the number of the day within the week, the name of the month, etc. Each attribute is stored as a column. The query below adds an INSERT statement to the WHILE loop.

```
WHILE @currentDate < @EndDate
BEGIN
-- For each date in the range, we're going to insert a row into our DimDate table.
	INSERT INTO dbo.Dates (DateValue,YearofDate,QuarterofDate,MonthofYear,

	NameofMonth,DayOfMonth,NameofDay,DayofWeek,DayofYear,WeekofYear)
	VALUES
		(
			@currentDate
			,year(@currentDate)
			,DATEPART(q,@currentDate)
			,DATEPART(m,@currentDate)
			,DATENAME(m,@currentDate)
			,DATEPART(d,@currentDate)
			,DATENAME(WEEKDAY,@currentDate)
			,DATEPART(dw,@currentDate)
			,DATEPART(dy,@currentDate)
			,DATEPART(wk,@currentDate)
		)
-- Now that the row is inserted, we need to increment @currentDate by one day
-- and process the next row.
	set @currentDate = DATEADD(day,1,@currentDate);

END
```

Below is the complete script to create a simple date table:

```
drop table if exists dbo.Dates;

create table dbo.Dates
(
	DateValue date,
	YearOfDate int,
	QuarterOfDate int,
	MonthOfYear int,
	NameOfMonth varchar(15),
	DayOfMonth int,
	NameOfDay varchar(15),
	DayOfWeek int,
	DayOfYear int,
	WeekOfYear int
)


DECLARE @StartDate date = '01/01/1950';
DECLARE @EndDate date = '01/01/2050';
DECLARE @currentDate date = @StartDate;

WHILE @currentDate < @EndDate
BEGIN
-- For each date in the range, we're going to insert a row into our DimDate table.
	INSERT INTO dbo.Dates (DateValue,YearofDate,QuarterofDate,MonthofYear,

	NameofMonth,DayOfMonth,NameofDay,DayofWeek,DayofYear,WeekofYear)
	VALUES
		(
			@currentDate
			,year(@currentDate)
```

```
                    ,DATEPART(q,@currentDate)
                    ,DATEPART(m,@currentDate)
                    ,DATENAME(m,@currentDate)
                    ,DATEPART(d,@currentDate)
                    ,DATENAME(WEEKDAY,@currentDate)
                    ,DATEPART(dw,@currentDate)
                    ,DATEPART(dy,@currentDate)
                    ,DATEPART(wk,@currentDate)
                )
-- Now that the row is inserted, we need to increment @currentDate by one day
-- and process the next row.
       set @currentDate = DATEADD(day,1,@currentDate);

END
```

Creating a Star Schema

The ***star schema*** is a name given to a specific way of organizing data in tables for purpose of analysis. It is an implementation of a ***dimensional model***. A dimensional model is a way of structuring quantitative data so that it represents a real-world business or organizational process.

To create a star schema, we first need to identify those values from our data that give us primary information about our business process. We could call these values our primary phenomena of interest. In the CarDealer database, for example, we could sum up the SalesAmount in the Sales table to calculate Revenue. We could count the number of cars sold. We could subtract the DealerCost of the car from the SalesAmount and calculate profit.

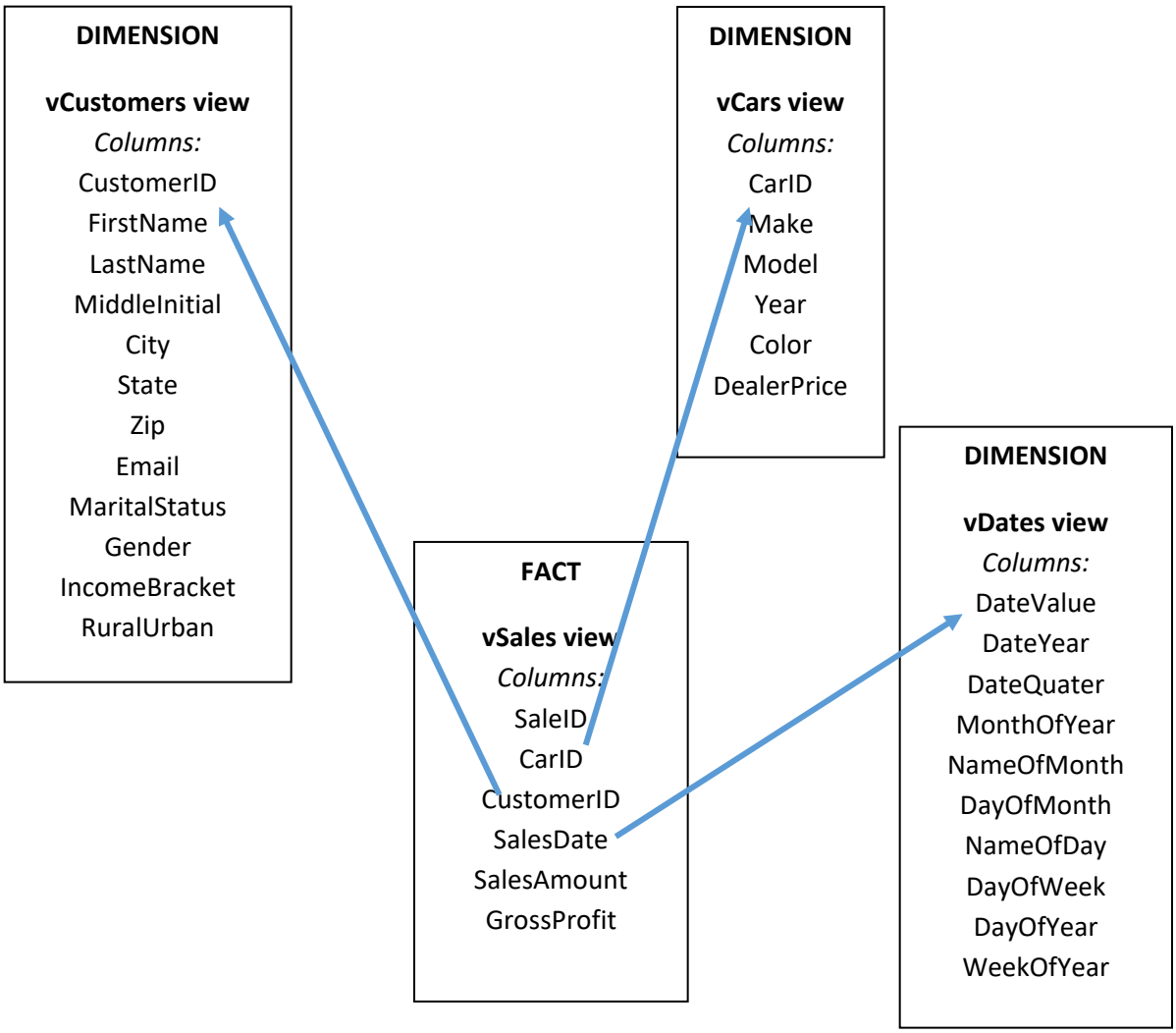
Revenue, number of cars sold, and profit would all be things of primary interest to us. The likely goal of any analysis we perform is to understand all the things that i*nfluence* these values. We call values such as revenue, profit, number of cars sold ***measures***. In our star schema, measures are calculated from ***fact columns*** in a ***fact table***. In addition to fact columns, the fact table contains foreign key columns pointing to rows in ***dimension tables***.

All of those things that might impact the value of fact columns, and therefore measure, are organized by ***entity*** into dimension tables. In CarDealer, for example, customers represent an entity that comes to bear on our measures. In other words, customers have attributes such as an income bracket, a marital status, a geographic region in which they live, and many other things we might want to assess in relation to our measures.

We'll create a ***dimension table*** to store all of our information about customers. Each row would represent an individual customer, and each column would be some attribute of that customer we think has meaning in the analysis. In addition to customers, we'll create dimension tables for cars and dates.

We're going to implement our star schema as a collection of views plus our date table. As we'll see in the last module, this provides a simple and efficient method of preparing data for analytical tools such as Power BI. Specifically, we'll create a view named **Sales** to represent a fact table, we'll create views to

represent car and customer dimension tables, and we can query our date table directly. This will create the following star schema:



The below TSQL code will create our four views:

```
drop view if exists vCars;
go
create view vCars
as
select
       c.CarID
       ,c.Make
       ,c.Model
       ,c.Year
       ,c.Color
from dbo.Cars c;

go
drop view if exists vSales;
go
create view vSales
as
select
       s.SaleID
       ,s.CarID
       ,s.CustomerID
       ,s.SalesDate
       ,s.SalesAmount
       ,c.DealerPrice
       ,s.SalesAmount - c.DealerPrice as [GrossProfit]
from dbo.Sales s
join dbo.Cars c
on s.CarID=c.CarID;

go

drop view if exists vCustomers
go
create view vCustomers
as
select
       c.CustomerID
       ,p.FirstName
       ,p.LastName
       ,p.MiddleInitial
       ,p.City
       ,p.State
       ,p.Zip
       ,p.Email
       ,cd.MaritalStatus
       ,cd.Gender
       ,cd.IncomeBracket
       ,cd.RuralUrban
from dbo.Customers c
join dbo.Person p
on c.PersonID=p.PersonID
join dbo.CustomerDemographics cd
on c.DemographicsGroup=cd.DemogroupID;

go

create view vDates
as
```

```
select
       Date as [DateValue]
       ,Year as [DateYear]
       ,Quarter as [DateQuater]
       ,Month as [MonthOfYear]
       ,MonthName as [NameOfMonth]
       ,DayOfMonth
       ,DayName as [NameOfDay]
       ,DayOfWeekUSA as [DayOfWeek]
       ,DayOfYear
       ,WeekOfYear
from dbo.DimDate;
```

Another thing to understand about the star schema is the direction across a relationship in which aggregations will generally be made. Our fact table, Sales, contains columns that will be aggregated, and those aggregations will be compared across ***grouping factors*** from the dimension tables.

An analysis we'd likely want to do with CarDealer is compare Revenue across car makes, models, and years. Revenue is a measure, and we'll calculate it by summing across individual sales records. But we'll do it within make, model, and year groups. The same will usually be true with columns from vCars and Dates. For example, we might want to compare the sum of gross profit across make, model, year and the month and year the car sold.

Again, it's the fact table, sales, that contains the values of primary interest, such as sales and profit. The dimension tables all contain columns that represent things we think might impact or otherwise be related to our values of primary interest, such as car makes, customer cities, and months of the year.

Case Table vs Star Schema

When we produce a single table of data for use in an analysis, it's usually in a ***case table*** format. This simply means that each row represents an instance, or case, of the thing you have information about, and each column represents an attribute of the thing. The following query would produce a case table, for example:

```
select
       c.Make
       ,c.Model
       ,c.Year
       ,s.SalesAmount
from dbo.Sales s
join dbo.Cars c
on s.CarID=c.CarID;
```

Each row, or case, represents an individual sale, and each column represents an attribute of the sale. We could easily analyze this in an Excel pivot table or a Python data frame.

One of the benefits of the star schema is the ease with which case tables can be created from it. The following query shoes that a join of the fact table to each dimension table will return the entire data set as a single case table:

```
select
       c.*
       ,cust.*
       ,d.*
       ,s.*
from dbo.vSales s
join dbo.vCars c
on s.CarID=c.CarID
join vCustomers cust
on s.CustomerID=cust.CustomerID
join Dates d
on s.SalesDate=d.DateValue;
```

As the data set gets larger, the star schema becomes the more efficient method of organizing and eventually (i.e., in client tools such as Power BI) storing and retrieving the data.

Additional Reading

Understanding star schema and the importance for Power BI. Microsoft documentation providing a conceptual overview of the star schema, and its importance for analytical tools such as Power BI. https://docs.microsoft.com/en-us/power-bi/guidance/star-schema.

Views. Microsoft documentation on views. https://docs.microsoft.com/en-us/sql/relational-databases/views/views?view=sql-server-ver16

Create Procedure. Microsoft documentation on the CREATE PROCEDURE statement. https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-ver16

Create Function. Microsoft documentation for CREATE FUNCTION. https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-ver16