**This video explains this document and walks you through the process.**

[**https://vimeo.com/229389216**](https://vimeo.com/229389216)

**pw: drive2020**

**Model**

The star schema separates business process data into facts, which hold the measurable, quantitative data about a business, and dimensions which are descriptive attributes related to fact data. Examples of fact data include sales price, sale quantity, and time, distance, speed, and weight measurements. Related dimension attribute examples include product models, product colors, product sizes, geographic locations, and salesperson names.

A star schema that has many dimensions is sometimes called a *centipede schema*.[[2]](http://en.wikipedia.org/wiki/Star_schema#cite_note-Kimball-DWHToolkit-2) Having dimensions of only a few attributes, while simpler to maintain, results in queries with many table joins and makes the star schema less easy to use.

**Fact tables**

Fact tables record measurements or metrics for a specific event. Fact tables generally consist of numeric values, and foreign keys to dimensional data where descriptive information is kept.[[2]](http://en.wikipedia.org/wiki/Star_schema#cite_note-Kimball-DWHToolkit-2)Fact tables are designed to a low level of uniform detail (referred to as "granularity" or "grain"), meaning facts can record events at a very atomic level. This can result in the accumulation of a large number of records in a fact table over time. Fact tables are defined as one of three types:

* Transaction fact tables record facts about a specific event (e.g., sales events)
* Snapshot fact tables record facts at a given point in time (e.g., account details at month end)
* Accumulating snapshot tables record aggregate facts at a given point in time (e.g., total month-to-date sales for a product)

Fact tables are generally assigned a [surrogate key](http://en.wikipedia.org/wiki/Surrogate_key) to ensure each row can be uniquely identified.

**Dimension tables**

Dimension tables usually have a relatively small number of records compared to fact tables, but each record may have a very large number of attributes to describe the fact data. Dimensions can define a wide variety of characteristics, but some of the most common attributes defined by dimension tables include:

* Time dimension tables describe time at the lowest level of time granularity for which events are recorded in the star schema
* Geography dimension tables describe location data, such as country, state, or city
* Product dimension tables describe products
* Employee dimension tables describe employees, such as salespeople
* Range dimension tables describe ranges of time, dollar values, or other measurable quantities to simplify reporting

Dimension tables are generally assigned a [surrogate primary key](http://en.wikipedia.org/wiki/Surrogate_key), usually a single-column integer data type, mapped to the combination of dimension attributes that form the natural key.

**Benefits**

Star schemas are [denormalized](http://en.wikipedia.org/wiki/Database_normalization), meaning the normal rules of [normalization](http://en.wikipedia.org/wiki/Database_normalization) applied to transactional relational databases are relaxed during star schema design and implementation. The benefits of star schema [denormalization](http://en.wikipedia.org/wiki/Database_normalization) are:

* Simpler queries - star schema join logic is generally simpler than the join logic required to retrieve data from a highly [normalized](http://en.wikipedia.org/wiki/Database_normalization) transactional schemas.
* Simplified business reporting logic - when compared to highly [normalized](http://en.wikipedia.org/wiki/Database_normalization) schemas, the star schema simplifies common business reporting logic, such as period-over-period and as-of reporting.
* Query performance gains - star schemas can provide performance enhancements for read-only reporting applications when compared to highly [normalized](http://en.wikipedia.org/wiki/Database_normalization) schemas.

Fast aggregations - the simpler queries against a star schema can result in improved performance for aggregation operations.

**Disadvantages**

The main disadvantage of the star schema is that [data integrity](http://en.wikipedia.org/wiki/Data_integrity) is not enforced as well as it is in a highly [normalized](http://en.wikipedia.org/wiki/Database_normalization) database. One-off inserts and updates can result in data anomalies which [normalized](http://en.wikipedia.org/wiki/Database_normalization) schemas are designed to avoid. Generally speaking, star schemas are loaded in a highly controlled fashion via batch processing or near-real time "trickle feeds", to compensate for the lack of protection afforded by [normalization](http://en.wikipedia.org/wiki/Database_normalization).

**STAR & SNOWFLAKE**

In Datawarehousing concept Dimension modeling takes very important place. There are two methods for Dimension modeling.  
  
In STAR schema all the facts stored in one central table and using Primary key and foreign key relationship other Dimension tables are connected with fact table.  
  
SnowFlake schema is a structure where we can have more than one fact table or Dimension tables are aggregated in higher level. Generaly used to avoid complixity( easy to understand) and Create more normalize structure.

*In terms of Optimization or fast retrieval of data, STAR is recommended, because of less travel in dynamic query.*

I. **How to Build a Data Model Using Star Schema**

a. Determine the Facts and Dimensions

b. Denormalize the data

c. Create Dimension Tables

d. Create Fact Table

II. **Determine the Facts and Dimensions**

a. Dimensions

1. Territory

a. TerritoryName

b. TerritoryGroup

2. Time

a. OrderDate

b. OrderMonth

c. OrderQuarter

d. OrderYear

3. Product

a. ProductName

b. ProductSubcategory

c. ProductCategory

b. Fact

1. Sales

a. OrderQty

b. UnitPrice

c. UnitPriceDiscount

d. LineTotal

III. **Denormalize the Data** - Denormalization is the process of attempting to optimize the read performance of a database by adding redundant data or by grouping data. Basically denormalization is what you get when you join multiple tables together. It resembles an Excel Spreadsheet*.*

*If you are not working with a relational database, more than likely your data is already denormalized.*

a. **Connect to the Database** - Open your local instance of SQL Server and connect to the AdventureWorks database.

b. **Joining the Data** - D*enormalized* the Data by creating one big query joining all of the data and storing it in a temp table.

SELECT d.ProductID,d.Name ProductName,e.Name ProductSubcategory,

f.Name ProductCategory, --Product Dimension Columns

b.OrderDate,MONTH(b.OrderDate) OrderMonth,DATEPART(QQ,b.OrderDate) OrderQuarter,YEAR(b.OrderDate) OrderYear, --Time Dimension Columns

c.TerritoryID,c.Name TerritoryName,c.[Group] TerritoryGroup, --Territory Dimension Columns

a.SalesOrderDetailID,a.OrderQty,a.UnitPrice,a.UnitPriceDiscount,a.LineTotal

-- Sales Facts

INTO #SalesDataModel

FROM Sales.SalesOrderDetail a

LEFT JOIN [Sales].[SalesOrderHeader] b

ON a.SalesOrderID = b.SalesOrderID

LEFT JOIN Sales.SalesTerritory c

ON b.TerritoryID = c.TerritoryID

LEFT JOIN Production.Product d

ON a.ProductID = d.ProductID

LEFT JOIN Production.ProductSubcategory e

ON d.ProductSubcategoryID = e.ProductSubcategoryID

LEFT JOIN Production.ProductCategory f

ON e.ProductCategoryID = f.ProductCategoryID

c. **Explanation** - By denormalizing the data, we insure our data is all related. Also it’s easier to create dimension and facts from one big dataset.

IV. **Creating Dimensions**

a**. Create Table** - Create a table if it doesn’t Exist

1. Make sure the table has an Identity Column for the Primary Key.

2. Checksum – This is a function used to return a calculated sum on a row of data. Lets say the check sum for a column is 22222222. When comparing the #SalesDataModel with the Product Dimension, this computed number will let us know if data in the row has been altered. If anything changes in the row, the CheckSum will produce a different number. This number will be compared with the existing records in the Product Dimension and only if the number has changes, will the row be updated.

IF NOT EXISTS (SELECT \* FROM SYSOBJECTS where name='Dim\_Product' and xtype='U') -- Creates the Dim\_Product Table if Exist

BEGIN

CREATE TABLE Dim\_Product (

ProductKey INT IDENTITY(1,1),

ProductID INT,

ProductName VARCHAR(200),

ProductSubcategory VARCHAR(200),

ProductCategory VARCHAR(200),

ProductCheckSum int

)

END

b. **Merge Statement** – MERGE is a new feature in 2008 that provides an efficient way to perform multiple DML operations. In previous versions of SQL Server, we had to write separate statements to INSERT, UPDATE, or DELETE data based on certain conditions, but now, using MERGE statement we can include the logic of such data modifications in one statement that even checks when the data is matched then just update it and when unmatched then insert it. Lets create a MERGE statement to insert/update records in the Product Dimension.

MERGE Dim\_Product AS TARGET

USING ( SELECT DISTINCT ProductID,ProductName,ProductSubcategory,ProductCategory,

BINARY\_CHECKSUM(ProductName,ProductSubcategory,ProductCategory) ProductCheckSum

FROM #SalesDataModel

) AS SOURCE

ON TARGET.ProductID = SOURCE.ProductID

WHEN MATCHED and TARGET.ProductCheckSum <> SOURCE.ProductCheckSum

THEN

UPDATE

SET TARGET.ProductName = SOURCE.ProductName,

TARGET.ProductSubcategory = SOURCE.ProductSubcategory,

TARGET.ProductCategory = SOURCE.ProductCategory

WHEN NOT MATCHED

THEN

INSERT(ProductID,ProductName,ProductSubcategory, ProductCategory,ProductCheckSum)

VALUES (SOURCE.ProductID,SOURCE.ProductName, SOURCE.ProductSubcategory, SOURCE.ProductCategory,BINARY\_CHECKSUM(SOURCE.ProductName, SOURCE.ProductSubcategory,SOURCE.ProductCategory));

*Note: Be sure to end the statement with a ;*

c. **Explanation** - Notice we are updating the Product Subcategory and Product Category if the Product exists. We are inserting the record if the Product doesn’t exist

c. **Slowly Changing Dimensions** – Dimensions that change over time are called Slowly Changing Dimensions. For instance, a Territory or Territory Group may change over time. SCD are often categorized into 3 types

1. **Type 1** – Overwriting old Values

a. **Benefits** – Not must thinking involved, just update the records with the new value. Our Product Dimension created earlier is an example of a Type 1 SCD

b. **Disadvantages** – No history will be kept. When the Product Subcategory or Category changes, old sales will not be reflected in the new names only.

2. **Type 2** – Creating another additional record – *The old values will not be replaced but a new row containing the new values will be added to the table*

a. **Benefits** – History can be kept. That means if a Territory moves to a different group, old sales will be reflected in the old group and new sales will be reflected in the newly changed group.

b. **Disadvantages** – Requires more thinking when creating and reporting since Sales must be calculated on the Territory Name/Group used at the time of the sale.

3. **Type 3** – Creating new fields – *The latest update to the changed values can be seen*

a. **Benefits** – Keeps track of the latest change

b. **Disadvantages** – If the entity in the Dimension is changed more than once, there is no way to retain the history.

4**. Creating a Type 2 SCD** - Lets create one with our Territory Dimension

a. **Create table if Exist** – Create the Territory Table if it exists

i. We will add a few extra columns to track our changes in the Type 2 SCD. TerritoryEffectiveDate / TerritoryEndDate / TerritoryCurrentRecord

IF NOT EXISTS (SELECT \* FROM SYSOBJECTS where name='Dim\_Territory' and xtype='U') -- Creates the Dim\_Product Table if Exist

BEGIN

CREATE TABLE Dim\_Territory (

TerritoryKey INT IDENTITY(1,1),

TerritoryID INT,

TerritoryName VARCHAR(200),

TerritoryGroup VARCHAR(200),

TerritoryCheckSum INT,

/\* TerritoryCheckSum column will store a calculation based on the records

in the table. This is how we will know if a record has changed. \*/

TerritoryEffectiveDate DATETIME,

TerritoryEndDate DATETIME,

TerritoryCurrentRecord VARCHAR(50)

)

END

b. **Merge Statement with Type 2 SCD** – Let’s create a Merge Statement with a Type 2 SCD on the Territory Dimension.

INSERT INTO Dim\_Territory (TerritoryID,TerritoryName,TerritoryGroup,TerritoryCheckSum,TerritoryEffectiveDate,TerritoryEndDate,TerritoryCurrentRecord)

SELECT TerritoryID,

TerritoryName,

TerritoryGroup,

TerritoryCheckSum,

TerritoryEffectiveDate,

TerritoryEndDate,

TerritoryCurrentRecord

FROM

(

MERGE Dim\_Territory AS TARGET

USING (SELECT DISTINCT TerritoryID,TerritoryName,TerritoryGroup, BINARY\_CHECKSUM(TerritoryName, TerritoryGroup) TerritoryCheckSum

FROM #SalesDataModel

) SOURCE

ON TARGET.TerritoryID = SOURCE.TerritoryID

WHEN MATCHED and TARGET.TerritoryCheckSum <> SOURCE.TerritoryCheckSum

THEN

UPDATE

SET TARGET.TerritoryEndDate = GETDATE(),

TARGET.TerritoryCurrentRecord = 'N'

WHEN NOT MATCHED

THEN

INSERT (TerritoryID,TerritoryName, TerritoryGroup, TerritoryEffectiveDate,TerritoryEndDate,TerritoryCheckSum,TerritoryCurrentRecord)

VALUES (SOURCE.TerritoryID,SOURCE.TerritoryName, SOURCE.TerritoryGroup,GETDATE(),'12/31/9999',

BINARY\_CHECKSUM(SOURCE.TerritoryName, SOURCE.TerritoryGroup),'Y')

/\*

1/1/2000 is entered for the TerritoryEffective Date only because we are dealing with past data. Normally this would be a GetDate()

\*/

OUTPUT $action,

SOURCE.TerritoryID,

SOURCE.TerritoryName,

SOURCE.TerritoryGroup,

SOURCE.TerritoryCheckSum,

'1/1/2000',

'12/31/9999',

'Y'

) Changes

(

Action,

TerritoryID,

TerritoryName,

TerritoryGroup,

TerritoryCheckSum,

TerritoryEffectiveDate,

TerritoryEndDate,

TerritoryCurrentRecord

)

WHERE Action = 'UPDATE'

c. **Testing the Type 2 SCD**

1. Run the code to create the dimension

2. Right Click on the *Sales.SalesTerritory* Table and *edit top 200* rows

2. Change the Territory Group for the *NorthWest* Territory to *Europe*

3. Run the code again and *Select \* from Dim\_Territory*

a. Notice the old NorthWest Territory had it’s TerritoryCurrentRecord column updated to ‘N’ as well as the TerritoryEndDate has been updated.

b. The new NorthWest territory that was switched to Europe is the new active record.

4. Change the Territory Group for NorthWest back to North America

5. Run the code again and *Select \* from Dim\_Territory*

6. Notice that changing the Territory Name back created another record in the Type 2 SCD

d. **Explanation** – Even though the Type 2 SCD takes more code to create, it is very effective as it will keep track of all changes in the Dimension.

5. Create the final Time Dimension using a Type 1 SCD. The time dimension will not change over time so there is no need to set this up as a Type 2 or 3 SCD.

IF NOT EXISTS (SELECT \* FROM SYSOBJECTS where name='Dim\_Time' and xtype='U') -- Creates the Dim\_Product Table if Exist

BEGIN

CREATE TABLE Dim\_Time (

TimeKey INT IDENTITY(1,1),

OrderDate DATETIME,

OrderMonth INT,

OrderQuarter INT,

OrderYear INT,

TimeCheckSum int

)

END

MERGE Dim\_Time AS TARGET

USING ( SELECT DISTINCT OrderDate,OrderMonth,OrderQuarter,OrderYear,

BINARY\_CHECKSUM(OrderMonth,OrderQuarter,OrderYear) TimeCheckSum

FROM #SalesDataModel

) AS SOURCE

ON TARGET.OrderDate = SOURCE.OrderDate

WHEN MATCHED and TARGET.TimeCheckSum <> SOURCE.TimeCheckSum

THEN

UPDATE

SET TARGET.OrderMonth = SOURCE.OrderMonth,

TARGET.OrderQuarter = SOURCE.OrderQuarter,

TARGET.OrderYear = SOURCE.OrderYear

WHEN NOT MATCHED

THEN

INSERT (OrderDate,OrderMonth,OrderQuarter,OrderYear,TimeCheckSum)

VALUES (SOURCE.OrderDate,SOURCE.OrderMonth, SOURCE.OrderQuarter,SOURCE.OrderYear,BINARY\_CHECKSUM( SOURCE.OrderMonth,SOURCE.OrderQuarter,SOURCE.OrderYear));

V. **Creating Fact Table**

a**. Create Fact Table** - Create a table if it doesn’t Exists. The fact table will be based on the #SalesDataModel temp table; however, instead of using all of the columns in the Product / Territory / Time Dimension, we will use the surrogate keys created in the Dimension tables as well as the Facts. The idea is for the Fact table, which will be your largest table in the data model, to have nothing but number columns. SQL queries numbers with better performance than strings. Also for the Territory Dimension, we must add an extra clause in our join to account for the Type 2 SCD.

IF NOT EXISTS (SELECT \* FROM SYSOBJECTS where name='Fact\_Sales' and xtype='U') -- Creates the Dim\_Product Table if Exist

BEGIN

CREATE TABLE Fact\_Sales (

FactSalesKey INT IDENTITY(1,1),

SalesOrderDetailID INT,

ProductKey INT,

TerritoryKey INT,

TimeKey INT,

OrderQty INT,

UnitPrice MONEY,

UnitPriceDiscount MONEY,

LineTotal MONEY,

FactCheckSum INT

)

END

b. **Populating the Fact Table** - We will create a Merge statement to update/insert records in the fact table.

MERGE Fact\_Sales AS TARGET

USING ( SELECT a.SalesOrderDetailID,b.ProductKey,c.TerritoryKey, d.TimeKey, OrderQty,UnitPrice,UnitPriceDiscount ,LineTotal,BINARY\_CHECKSUM(ProductKey,TerritoryKey, TimeKey,OrderQty,UnitPrice,UnitPriceDiscount,LineTotal) FactCheckSum

FROM #SalesDataModel a

LEFT JOIN Dim\_Product b

ON a.ProductID = b.ProductID

LEFT JOIN Dim\_Territory c

ON a.TerritoryID = c.TerritoryID

AND a.OrderDate between c.TerritoryEffectiveDate AND c.TerritoryEndDate

LEFT JOIN Dim\_Time d

ON a.OrderDate = d.OrderDate

) AS SOURCE

ON TARGET.SalesOrderDetailID = SOURCE.SalesOrderDetailID

WHEN MATCHED and TARGET.FactCheckSum <> SOURCE.FactCheckSum

THEN

UPDATE

SET TARGET.ProductKey = SOURCE.ProductKey,

TARGET.TerritoryKey = SOURCE.TerritoryKey,

TARGET.TimeKey = SOURCE.TimeKey,

TARGET.OrderQty = SOURCE.OrderQty,

TARGET.UnitPrice = SOURCE.UnitPrice,

TARGET.UnitPriceDiscount = SOURCE.UnitPriceDiscount,

TARGET.LineTotal = SOURCE.LineTotal

TARGET.FactCheckSum = SOURCE.FactCheckSum

WHEN NOT MATCHED

THEN

INSERT( SalesOrderDetailID,ProductKey,TerritoryKey,TimeKey, OrderQty,UnitPrice,UnitPriceDiscount,LineTotal,FactCheckSum)

VALUES (SOURCE.SalesOrderDetailID,SOURCE.ProductKey, SOURCE.TerritoryKey,SOURCE.TimeKey,SOURCE.OrderQty,

SOURCE.UnitPrice,SOURCE.UnitPriceDiscount,SOURCE.LineTotal,

BINARY\_CHECKSUM(SOURCE.ProductKey, SOURCE.TerritoryKey,SOURCE.TimeKey,SOURCE.OrderQty, SOURCE.UnitPrice, SOURCE.UnitPriceDiscount,SOURCE.LineTotal));

V. **Creating the Stored Procedure**

a**. Create Final SP** – Create a stored procedure (sp\_DW\_Update) to run all of the code created. This stored procedure would normally have an automated job setup to update the DW.

