|  |
| --- |
| **National University of Computer and Emerging Sciences** |
| Lab Manual 10  “**Partitioning Techniques and Indexed Views**” |
|  |
| Data Warehousing and Data Mining |
|  |
|  |
|  |

|  |  |
| --- | --- |
| Course Instructor | Sir Ishaq Raza |
| Lab Instructor(s) | Farhan Azhar Nouman Ali |
| Section | CS |
| Semester | Fall 2019 |

|  |
| --- |
|  |

# **Partition Cube**

**Partition Benefits:**

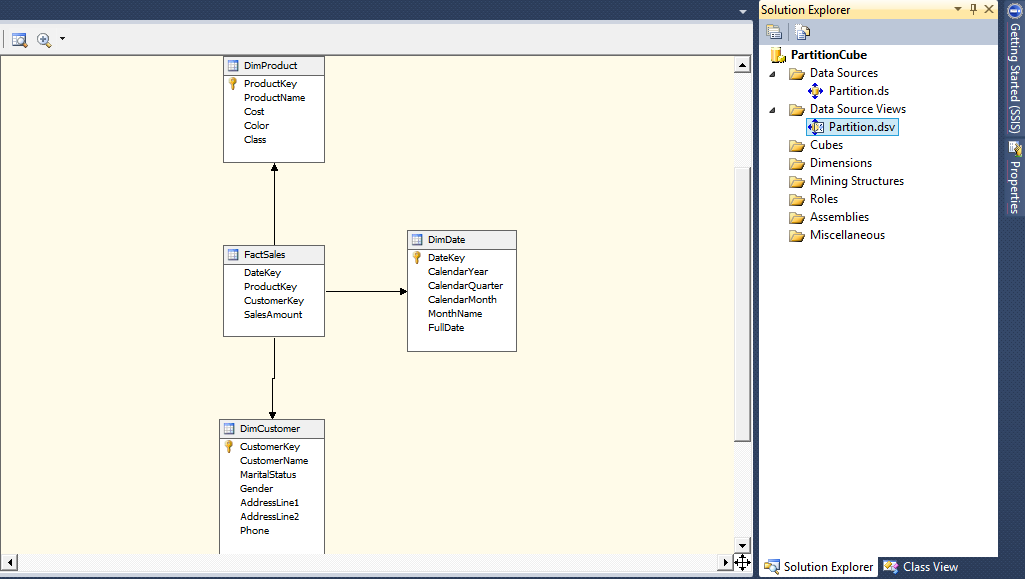
* **Better Query Performance**: Cube partition is a powerful mechanism for improving query performance. Queries that summarize data over 10 years could take considerably longer than those that only search through the current year data. If we have proper partitions then SSAS only has to scan a small subset of data to return query results hence dramatic performance improvements compared to queries running against a cube with a single partition.
* **Minimize downtime**:  Cube partitioning supports reducing downtime associated with cube processing. In almost all the cases, a portion of data warehouse is volatile and needs to be processed often. However other portions are relatively static. For example, in a sales cube, we need to change the current year's data nightly, but sales from previous years might change only occasionally - in case if account for merchandise returns and exchanges. If your warehouse tracks last 10 years sales then processing only the current partition may be 10 times quicker than processing the entire cube.
* **Aggregations benefits**: The partition queried frequently could benefit from additional aggregations, which in turn could improve performance. Partition(s) that are used less can be processed less frequently with considerably fewer aggregations.
* **Customized storage and processing settings**: Frequently accessed partitions might benefit from proactive caching and ROLAP storage. On the other hand, other forms of storage and processing might be better for less frequently queried partitions.
* **Distributed query and processing load**: SSAS allows you to create remote partitions - a remote partition resides on a server different from its parent cube. This way the queries that affect the remote partition are processed on a server separate from its parent cube, allowing you to take advantage of additional processing power.
* **Parallel Partitions Processing**: SSAS allows processing multiple partitions in parallel on a server that has multiple processors. This can further reduce the total cube processing time.

# **Implementing Partition Cube**

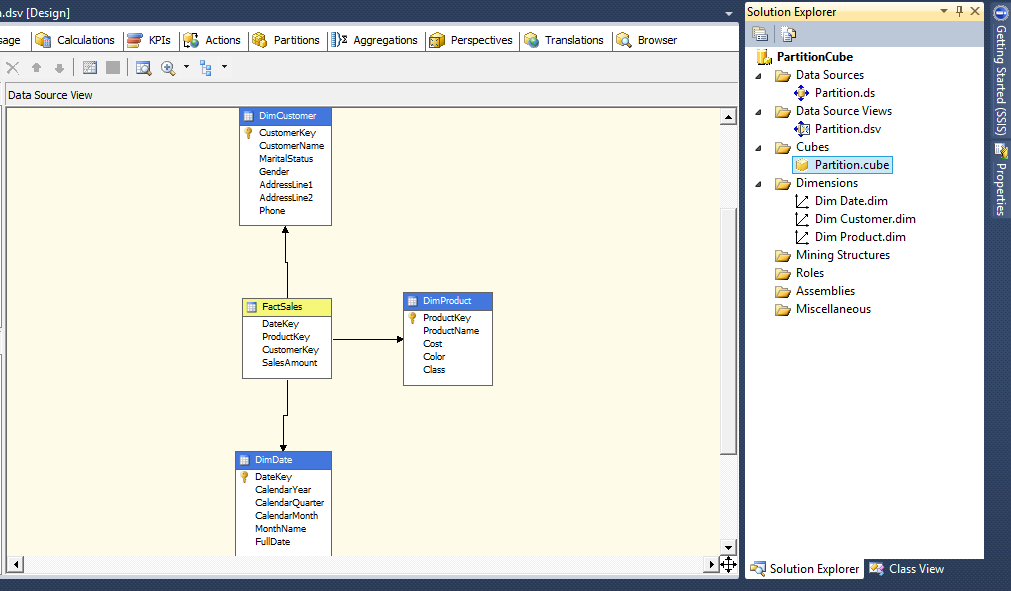
STEP1:   
Prepare test data for **Sales** cube. I will use three dimensions (DimDate, DimProduct, and DimCustomer) and one fact table for Sales cube. Run the Script file **PartitionCube** provided with the manual to make dimensions and fact table along with test data.

STEP2:   
Create new Analysis Services Project using **SQL server data tools** and save it as **PartitionCube**.

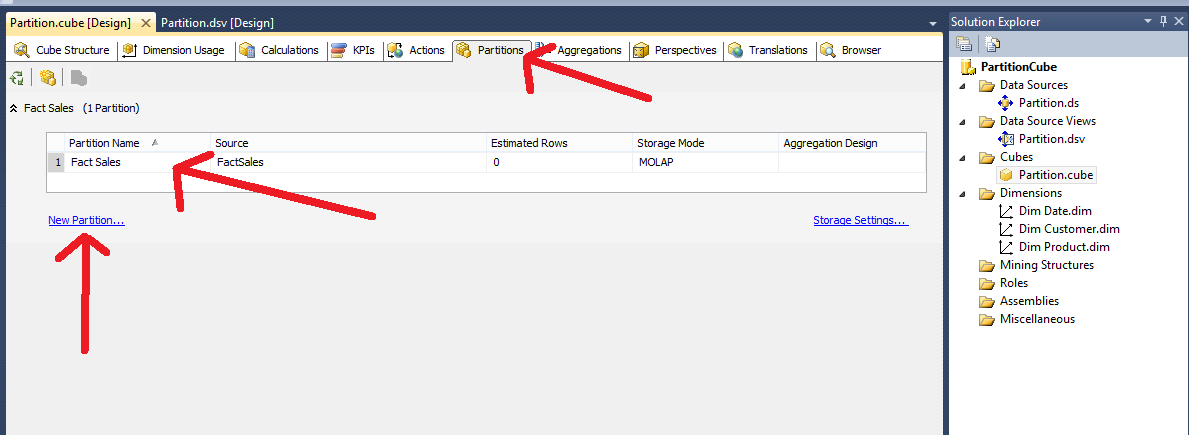
STEP3:   
Create **Data Source**, and then **Data Source View**as described in the last lab session.



STEP4:   
Create **cube** named **Partition** along with required dimensions.

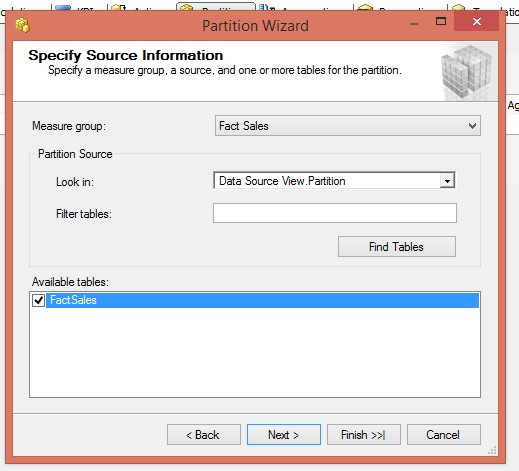


STEP5:   
Double click on the cube and navigate to **Partition** tab. You will see a default partition as shown below:

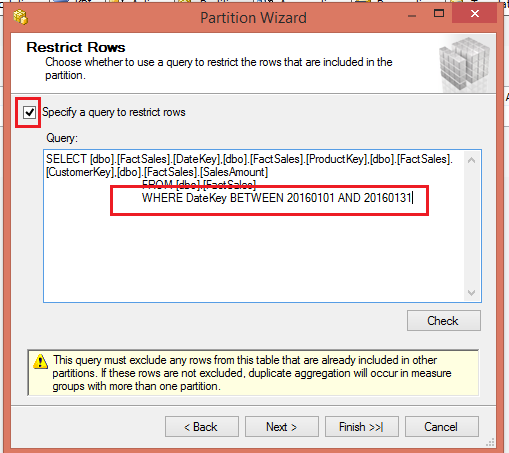


Delete this default partition and Click on **new partition...**.

In *Partition Wizard,*Select FactSales as available table in **Specify Source Information** page and click next.



Select **Specify a query to restrict row** in **Restrict Rows** page and write WHERE condition to restrict partition rows.



Click **Next** twice and enter Partition name **Sales\_20160101-20160131** in the **Completing the Wizard** page and click **Finish**.   
Finally **Process** the cube. Once cube is processed successfully, you can see a new database in Analysis Services

# **Indexed Views**

**Introduction to SQL Server indexed view**

Regular SQL Server views are the saved queries that provide some benefits such as query simplicity, business logic consistency, and security. However, they do not improve the underlying query performance.

Unlike regular views, **indexed views are materialized views** that stores data physically like a table hence may provide some the performance benefit if they are used appropriately.

To create an indexed view, you use the following steps:

1. First, create a view that uses the WITH SCHEMABINDING option which binds the view to the schema of the underlying tables.
2. Second, create a unique clustered index on the view. This materializes the view.

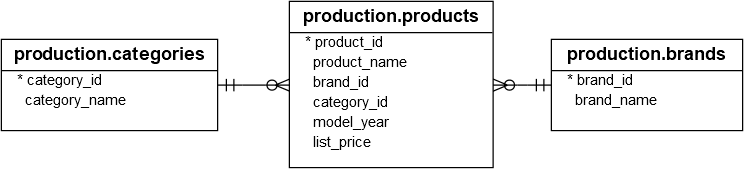
Because of the WITH SCHEMABINDING option, if you want to change the structure of the underlying tables which affect the indexed view’s definition, you must drop the indexed view first before applying the changes.

In addition, SQL Server requires all object references in an indexed view to include the two-part naming convention i.e., schema.object, and all referenced objects are in the same database.

When the data of the underlying tables changes, the data in the indexed view is also automatically updated. This causes a write overhead for the referenced tables. It means that when you write to the underlying table, SQL Server also has to write to the index of the view. Therefore, you should only create an indexed view against the tables that have in-frequent data updates.

## **Creating an SQL Server indexed view example**

The following statement creates an indexed view based on columns of the production.products, production.brands, and production.categories tables from the sample database provided



|  |
| --- |
| CREATE VIEW product\_master  WITH SCHEMABINDING  AS  SELECT      product\_id,      product\_name,      model\_year,      list\_price,      brand\_name,      category\_name  FROM      production.products p  INNER JOIN production.brands b      ON b.brand\_id = p.brand\_id  INNER JOIN production.categories c      ON c.category\_id = p.category\_id; |

Notice the option WITH SCHEMABINDING after the view name. The rest is the same as a regular view.

Before creating a unique clustered index for the view, let’s examine the query I/O cost statistics by querying data from a regular view and using the SET STATISTICS IO command:

|  |
| --- |
| SET STATISTICS IO ON  GO    SELECT      \*  FROM      product\_master  ORDER BY      product\_name;  GO |

SQL Server returns the following query I/O cost statistics:

|  |
| --- |
| Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.  Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.  Table 'products'. Scan count 1, logical reads 5, physical reads 1, read-ahead reads 3, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.  Table 'categories'. Scan count 1, logical reads 2, physical reads 1, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.  Table 'brands'. Scan count 1, logical reads 2, physical reads 1, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

As you can see clearly from the output, SQL Server had to read from three corresponding tables before returning the result set.

Let’s [add a unique clustered index](http://www.sqlservertutorial.net/sql-server-indexes/sql-server-unique-index/) to the view:

|  |
| --- |
| CREATE UNIQUE CLUSTERED INDEX      ucidx\_product\_id  ON product\_master(product\_id); |

This statement **materializes** the view, making it have a physical existence in the database.

You can also add a non-clustered index on the product\_name column of the view:

|  |  |
| --- | --- |
|  | CREATE NONCLUSTERED INDEX      ucidx\_product\_name  ON product\_master(product\_name); |

Now, if you query data against the view, you will notice that the statistics have changed:

|  |  |
| --- | --- |
|  | Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.  Table 'product\_master'. Scan count 1, logical reads 6, physical reads 1, read-ahead reads 11, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0. |

Instead of reading data from three tables, SQL Server now reads data directly from the materialized view product\_master.

You can also use the indexed view in another query and join it with other tables for faster results. For example the following query uses the indexed view to retrieve the records of production.products, production.brands, production.categories tables (indexed view was formed on these) and also the sales.order\_items table.

SELECT

\*

FROM

product\_master p WITH (NOEXPAND)

INNER JOIN sales.order\_items s

ON p.product\_id = s.product\_id

ORDER BY

product\_name;

You can observe in the statistics the view is used to read the three joined tables in one go:

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'order\_items'. Scan count 1, logical reads 25, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'product\_master'. Scan count 1, logical reads 6, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

**Note that this feature is only available on SQL Server Enterprise Edition.** If you use the SQL Server Standard or Developer Edition, you must use the WITH (NOEXPAND) table hint directly in the FROM clause of the query which you want to use the view like the following query:

|  |
| --- |
| SELECT      \*  FROM      product\_master WITH (NOEXPAND)  ORDER BY      product\_name; |

You have learned how to create a SQL Server indexed view defined against tables that have infrequent data updates to improve the query performance.

**Task on Indexed Views**

* Create indexed view on sales.customers, sales.orders and sales.order\_items tables from the schema, using the process described in tutorial.
* Retrieve data using this indexed view and report statistics on it.