SQL Server indexed views

March 17, 2016 by Ahmad Yaseen

SQL Server Views are virtual tables that are used to retrieve a set of data from one or more tables. The view's data is not stored in the database, but the real retrieval of data is from the source tables. When you call the view, the source table's definition is substituted in the main query and the execution will be like reading from these tables directly.

Views are mainly created for security purpose in order to restrict user access to specific columns i. These are also created for simplification purpose in order to encapsulate frequently executed, complex queries that read from multiple tables each time. Views perform multi-tables reading too, causing huge amount of IO operations. There are no performance benefits from using standard views; if the view definition contains complex processing and joins between huge numbers of rows from a combination of tables, and you are calling this view very frequently, performance degradation will be noticed clearly.

To enhance the performance of such complex queries, a unique clustered index can be created on the view, where the result set of that view will be stored in your database the same as a real table with a unique clustered index. The good thing here is – the queries that are using the table itself can benefits from the view's clustered index without calling the view itself. Maintaining the clustered index of the view to be unique, the data changed on the source table will be easily found and the change will be reflected to the view. Changing the data directly from the indexed view is possible but shouldn't be done. Also, it is possible to create non-clustered indexes on a view, providing more possibilities to enhance the queries calling the view.

You can benefit from indexed views if its data is not frequently updated, as the performance degradation of maintaining the data changes

of the indexed view is higher than the performance enhancement of using this Indexed View. Indexed views improve the performance of queries that use joins and aggregations in processing huge amount of data and are executed very frequently. The environments that are best suited to indexed views are data warehouses and the Online Analytical Processing (OLAP) databases. On the other hand, it will not improve the performance on tables with many writes and updates such as Online Transaction Processing (OLTP) databases.

Creating indexed views differs from creating normal views in that using the **SCHEMA BINDING** hint is not optional. This means that you will not be able to apply structure changes on the tables that may affect the indexed view unless you alter or drop that indexed view first. In addition, you need to specify two parts name of these tables including the schema with the table name in the view definition. Also, any user-defined function that is referenced by the created indexed view should be created using **WITH SCHEMABINDING** hint.

Once the Indexed view is created, its data will be stored in your database the same as any other clustered index, so the storage space for the view's clustered index should be taken into consideration. Having the indexed view's clustered index stored in the database, with its own statistics created to optimize the cardinality estimation, different from the underlying tables' statistics, the SQL engine will not waste the time substituting the source tables' definition in the main query, and it will read directly from the view's clustered index.

There are some limitations when you create an indexed view. You can't use **EXISTS**, **NOT EXISTS**, **OUTER JOIN**, **COUNT(*)**, **MIN**, **MAX**, **subqueries, table hints**, **TOP** and **UNION** in the definition of your indexed view. Also, it is not allowed to refer to other views and tables in other databases in the view definition. You can't use the **text**, **ntext**, **image** and **XML**, data types in your indexed views. **Float** data type can be used in the indexed view but can't be used in the clustered index. If the Indexed view's definition contains **GROUP BY** clause, you should add **COUNT_BIG(*)** to the view definition

Another restriction on creating an indexed view is that there are a few **SET** options that should have certain values in your database if you manage to create an indexed view in it. For example, the `ANSI_NULLS, ANSI_PADDING, ANSI_WARNINGS, ARITHABORT,CONCAT_NULL_YIELDS_NULL,` and `QUOTED_IDENTIFIER` options should be **ON**, and the `NUMERIC_ROUNDABORT` option should be **OFF**. Non-deterministic columns can't be used in the indexed view definition. These are the columns that don't return the same value each time, like the **GETDATE()** function.

Benefits of clustered indexes created for an indexed view depends on the SQL Server edition. If you are using SQL Server Enterprise edition, SQL Server Query Optimizer will automatically consider the created clustered index as an option in the execution plan if it is the best index found. Otherwise, it will use a better one. In the other SQL Server editions such as Standard edition, the SQL Server Query Optimizer will access all the underlying source tables and use its indexes. In order to force the SQL Server Query Optimizer to use the index view's clustered index in the execution plan for the query, you should use the **WITH (NOEXPAND)** table hint in the FROM clause.

Let's review a small demo to test and compare the performance of a standard view and an indexed view that will read the employee information required for his manager from the Employee, EmployeeDepartmentHistory, Department, Shift and EmployeePayHistory tables under the HumanResources schema from the SQLSHACKDEMO database.

The below script will create a standard view that retrieves the requested information:

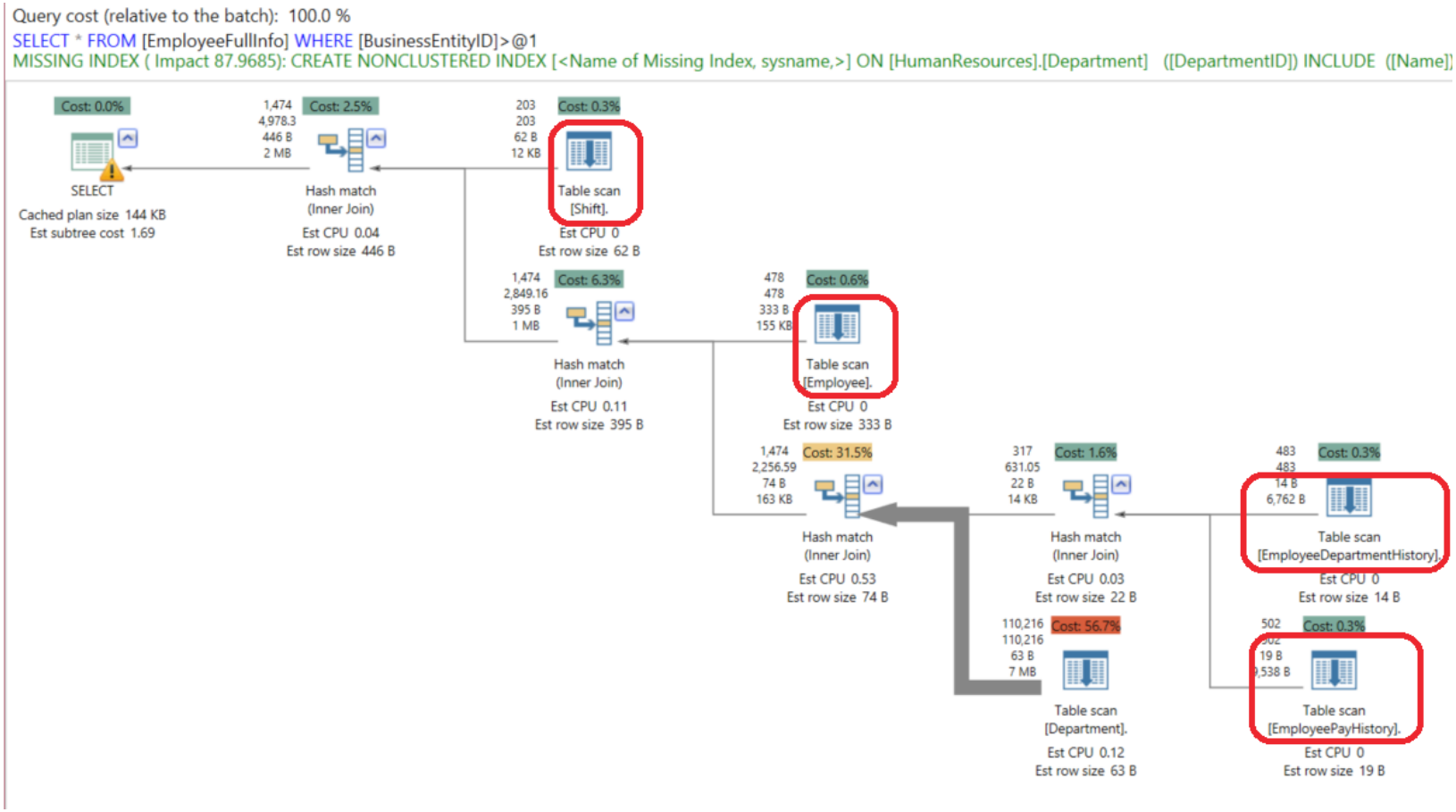
```
USE SQLShackDemo
GO
CREATE VIEW EmployeeFullInfo
AS
SELECT EMP.[BusinessEntityID]
      ,EMP.[LoginID]
      ,EMP.[JobTitle]
      ,EMP.[BirthDate]
      ,EMP.[MaritalStatus]
      ,EMP.[Gender]
      ,EMP.[HireDate]
      ,Dep.Name AS Department
      ,SH.Name AS ShiftName
      ,EMPPayHist.Rate AS EmployeeRate
  FROM [SQLShackDemo].[HumanResources].[Employee] AS EMP
  JOIN [SQLShackDemo].[HumanResources].[EmployeeDepartmentHistory] AS EMPDepHist
  ON EMP.BusinessEntityID =EMPDepHist.BusinessEntityID
  JOIN [SQLShackDemo].[HumanResources].[Department] AS Dep
  ON DEP.DepartmentID =EMPDepHist .DepartmentID
  JOIN [SQLShackDemo].[HumanResources].[Shift] SH
  ON EMPDepHist.ShiftID=SH.ShiftID
  JOIN [SQLShackDemo].[HumanResources].[EmployeePayHistory] EMPPayHist
  ON EMP.BusinessEntityID =EMPPayHist.BusinessEntityID
```

Once the EmployeeFullInfo view is created, the user's access is limited to see only the view columns and the complex logic that reads from the five tables is encapsulated into one small select statement from the view directly like the below one:

```
SELECT * FROM EmployeeFullInfo  where BusinessEntityID >12
```

As we can see from the query's execution plan generated using the APEXSQL Plan application, no performance benefits are achieved from this view, as the SQL Server Query Optimizer reads the data from the source tables performing Table Scan, ending with the below complex

plan.



To write an indexed view for the same previous complex logic, we need first to add the **WITH SCHEMABINDING** statement to the view as it is a must here. This will prevent any changes in the underlying tables that may affect the view's columns:

```
USE SQLShackDemo
GO
CREATE VIEW [HumanResources].EmployeeFullInfo_Indexed
WITH SCHEMABINDING AS
SELECT EMP.[BusinessEntityID]
      ,EMP.[LoginID]
      ,EMP.[JobTitle]
      ,EMP.[BirthDate]
```

```
      ,EMP.[MaritalStatus]
      ,EMP.[Gender]
      ,EMP.[HireDate]
      ,Dep.Name AS Department
      ,SH.Name AS ShiftName
      ,EMPPayHist.Rate AS EmployeeRate
  FROM [HumanResources].[Employee] AS EMP
  JOIN [HumanResources].[EmployeeDepartmentHistory] AS EMPDepHist
  ON EMP.BusinessEntityID =EMPDepHist.BusinessEntityID
  JOIN [HumanResources].[Department] AS Dep
  ON DEP.DepartmentID =EMPDepHist .DepartmentID
  JOIN [HumanResources].[Shift] SH
  ON EMPDepHist.ShiftID=SH.ShiftID
  JOIN [HumanResources].[EmployeePayHistory] EMPPayHist
  ON EMP.BusinessEntityID =EMPPayHist.BusinessEntityID
  GO
```

After creating the view, we will create a Unique Clustered Index on the EmployeeFullInfo_Indexed view covering all its fields:

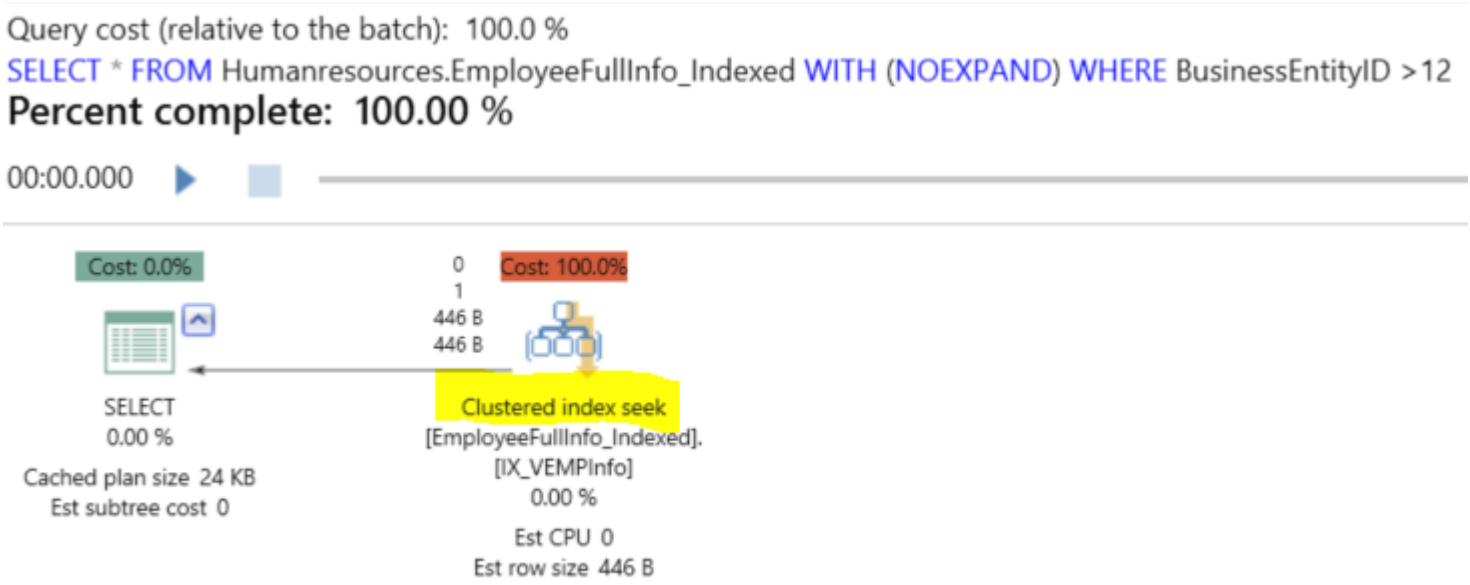
```
CREATE UNIQUE CLUSTERED INDEX IX_VEMPInfo
    ON EmployeeFullInfo_Indexed
     ([BusinessEntityID]
        ,[LoginID]
        ,[JobTitle]
        ,[BirthDate]
        ,[MaritalStatus]
        ,[Gender]
        ,[HireDate]
        ,Department
        ,ShiftName);
```

As the SQL Server edition installed on my test machine is standard edition, I need to force the SQL Server Query Optimizer to use the created index in the query plan by adding the WITH (NOEXPAND) table hint to my query as below:

```
SELECT * FROM EmployeeFullInfo_Indexed WITH (NOEXPAND) WHERE BusinessEntityID >12
```

As you can see from the execution plan generated using the APEXSQL Plan application, it looks totally different; rather than having the

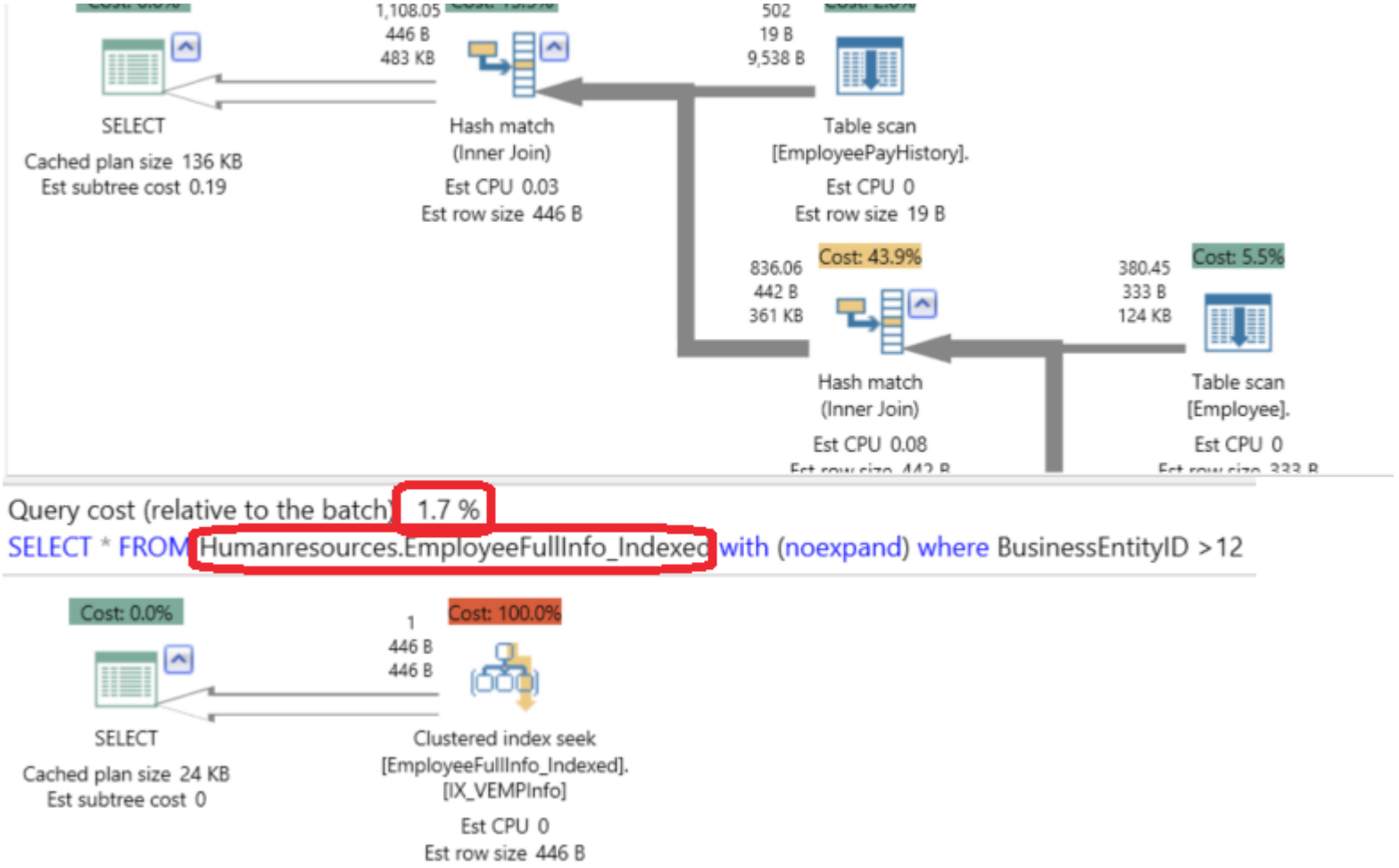
five tables' scan, the SQL Server Query Optimizer determines that using the view's clustered index is the optimal way to get the requested data from the view. It is clear that the optimizer reads all the data from the clustered index itself without touching the underlying tables.



In order to compare the performance of the two views and the enhancement we got from that indexed view, let's run the two SELECT statements in the same sessions, as follows, and study the cost shown in the execution plan for each one:

```
SELECT * FROM EmployeeFullInfo where BusinessEntityID >12
GO
SELECT * FROM EmployeeFullInfo_Indexed with (noexpand) where BusinessEntityID >12
```

The enhancement achieved by using the indexed view can be easily derived from the execution plan generated using the APEXSQL Plan application below, as the cost of using the standard view compared to the indexed view is 98:2, which means that the indexed view is better than the standard view by a factor of 20 times in our example:

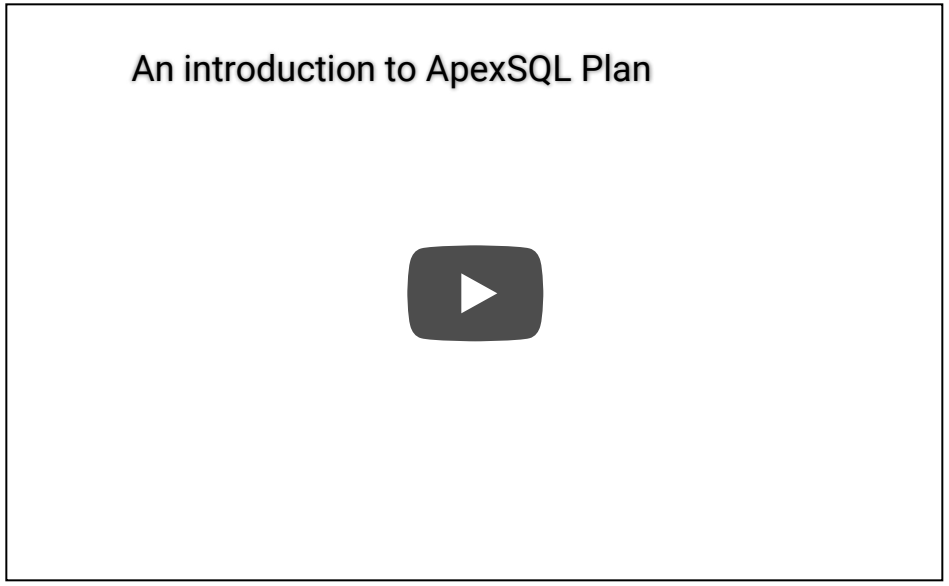


Conclusion

Using SQL Server indexed views can be considered as a good technique for enhancing query performance by reducing the IO cost and duration for the query, in addition to simplifying complex query logic when joining multiple tables and maintaining the data security. But it requires testing, planning, and deep studying why you need to use the indexed views and you should do a full analysis of the net performance impact, measuring performance enhancements vs costs

See more

Check out ApexSQL Plan, to view SQL execution plans for free, including comparing plans, stored procedure performance profiling, missing index details, lazy profiling, wait times, plan execution history and more



Ahmad Yaseen

Ahmad Yaseen is a SQL Server database administration leader at Aramex International Company with a bachelor's degree in computer engineering as well as .NET development experience.

He is a Microsoft Certified Professional with a good experience in SQL server development, administration, performance tuning, monitoring and high availability and disaster recovery technologies. Also, he is contributing with his SQL tips in many blogs

View all posts by Ahmad Yaseen

Related Posts:

1. SQL Server Partitioned Views
2. The SQL Server system views/tables/functions. Common questions and solutions to real life problems
3. SQL Server – development practices with referenced views
4. Discovering database specific information using built-in functions and dynamic management views (DMVs)
5. Restricting and monitoring SQL Server data access with SQL views and stored procedures

Views

39,156 Views

Name

Pankaj Sharma • 2 years ago

I have searched many links on the site and got if we create index view then it would be improve performance.
I just want to know, How unique clustered index work on schema binding view?

Regards
Rakesh kumar

6 | • Reply • Share ›

Ahmad Zuhair Yaseen ➚ Pankaj Sharma • 2 years ago

Thank you Sharma for your input here.

You can imagine it in the below way:

The view is a snapshot (virtual) of the main table that the user who reads from the view will read from.

Having a unique clustered index will make all pages that form (virtually) the view sorted , and the data within all these pages will be sorted, with unique values.

Searching from that sorted pages will be very fast, as there is no need to scan the underlying table or all view pages. SQL Server will directly seek for the value you are searching for.

Best Regards,
Ahmad

6 | • Reply • Share ›

erikj999 • 2 years ago

Hi Ahmed,

Do you know if the SQL 2016 SP1 license changes (CDC, columnstore, etc.) affected the optimizer behavior for Standard Edition with respect to indexed views? In other words, would I still need to use the NOEXPAND hint? I can't seem to find anything on line that

sheds light on that. Thanks!

4 ^ | v • Reply • Share ›

Ahmad Zuhair Yaseen ➚ erikj999 • 2 years ago

Thank you for your input here.

Yes you still need to use that hint.

Best Regards,
Ahmad

4 ^ | v • Reply • Share ›

Yesh v • 2 years ago

Hi. I have a SSRS report that is executing 4 SP's to pull the data and the reports is taking 10 minutes to pull the data. All the SP's are using views to pull the data and i am thinking that might be the issue. When i am trying to create indexed views it doesn't allow me to do and throwing error "Synonyms are invalid in a schemabound object or a constraint expression." Do you any other suggestions in improving the performance?. thanks

3 ^ | v • Reply • Share ›

Ahmad Zuhair Yaseen ➚ Yesh v • 2 years ago

Hello,
Did you get the chance to trace the execution plan for these SPs, as this will provide you with a suggested index, or minimum allows you to locate the heaviest operator that yo need to enhance.

Best Regards,
Ahmad

4 ^ | v • Reply • Share ›

Avatar

SQL replication with publisher database in Always on Availability Groups

Using Database Metrics to Predict Application Problems