Create Indexed Views

11/19/2018 • 10 minutes to read • +8

In this article

Applies to: ✓ SQL Server (all supported versions) ✓ Azure SQL Database

This article describes how to create indexes on a view. The first index created on a view must be a unique clustered index. After the unique clustered index has been created, you can create more nonclustered indexes. Creating a unique clustered index on a view improves query performance because the view is stored in the database in the same way a table with a clustered index is stored. The query optimizer may use indexed views to speed up the query execution. The view does not have to be referenced in the query for the optimizer to consider that view for a substitution.

Before You Begin

The following steps are required to create an indexed view and are critical to the successful implementation of the indexed view:

1. Verify the SET options are correct for all existing tables that will be referenced in the view.
2. Verify that the SET options for the session are set correctly before you create any tables and the view.
3. Verify that the view definition is deterministic.
4. Verify that the base table has the same owner as the view.
5. Create the view by using the `WITH SCHEMABINDING` option.
6. Create the unique clustered index on the view.

Important

When executing DML[1] on a table referenced by a large number of indexed views, or fewer but very complex indexed views, those referenced indexed views will have to be updated as well. As a result, DML query performance can degrade significantly, or in some cases, a query plan cannot even be produced. In such scenarios, test your DML queries before production use, analyze the query plan and tune/simplify the DML statement.

[1] Such as UPDATE, DELETE or INSERT operations.

Required SET Options for Indexed Views

Evaluating the same expression can produce different results in the Database Engine when different SET options are active when the query is executed. For example, after the SET option `CONCAT_NULL_YIELDS_NULL` is set to ON, the expression `'abc' + NULL` returns the value `NULL`. However, after `CONCAT_NULL_YIELDS_NULL` is set to OFF, the same expression produces `'abc'`.

To make sure that the views can be maintained correctly and return consistent results, indexed views require fixed values for several SET options. The SET options in the following table must be set to the values shown in the **Required Value** column whenever the following conditions occur:

- The view and subsequent indexes on the view are created.
- The base tables referenced in the view at the time the view is created.
- There is any insert, update, or delete operation performed on any table that participates in the indexed view. This requirement includes operations such as bulk copy, replication, and distributed queries.
- The indexed view is used by the query optimizer to produce the query plan.

SET options	Required value	Default server value	Default	Default
			OLE DB and ODBC value	**DB-Library value**

SET options	Required value	Default server value	Default	Default
			OLE DB and ODBC value	**DB-Library value**
ANSI_NULLS	ON	ON	ON	OFF
ANSI_PADDING	ON	ON	ON	OFF
ANSI_WARNINGS[1]	ON	ON	ON	OFF
ARITHABORT	ON	ON	OFF	OFF
CONCAT_NULL_YIELDS_NULL	ON	ON	ON	OFF
NUMERIC_ROUNDABORT	OFF	OFF	OFF	OFF
QUOTED_IDENTIFIER	ON	ON	ON	OFF

[1] Setting `ANSI_WARNINGS` to ON implicitly sets `ARITHABORT` to ON.

If you are using an OLE DB or ODBC server connection, the only value that must be modified is the `ARITHABORT` setting. All DB-Library values must be set correctly either at the server level by using **sp_configure** or from the application by using the SET command.

> ⓘ **Important**
>
> We strongly recommend that you set the `ARITHABORT` user option to ON server-wide as soon as the first indexed view or index on a computed column is created in any database on the server.

Deterministic Views

The definition of an indexed view must be deterministic. A view is deterministic if all expressions in the select list, as well as the `WHERE` and `GROUP BY` clauses, are deterministic. Deterministic expressions always return the same result any time they are evaluated with a specific set of input values. Only deterministic functions can participate in deterministic expressions. For example, the `DATEADD` function is deterministic because it always returns the same result for any given set of argument values for its three parameters. `GETDATE` is not deterministic because it is always invoked with the same argument, but the value it returns changes each time it is executed.

To determine whether a view column is deterministic, use the **IsDeterministic** property of the COLUMNPROPERTY function. To determine if a deterministic column in a view with schema binding is precise, use the **IsPrecise** property of the `COLUMNPROPERTY` function. `COLUMNPROPERTY` returns 1 if TRUE, 0 if FALSE, and NULL for input that is not valid. This means the column is not deterministic or not precise.

Even if an expression is deterministic, if it contains float expressions, the exact result may depend on the processor architecture or version of microcode. To ensure data integrity, such expressions can participate only as non-key columns of indexed views. Deterministic expressions that do not contain float expressions are called precise. Only precise deterministic expressions can participate in key columns and in `WHERE` or `GROUP BY` clauses of indexed views.

Additional Requirements

In addition to the SET options and deterministic function requirements, the following requirements must be met:

- The user that executes `CREATE INDEX` must be the owner of the view.
- When you create the index, the `IGNORE_DUP_KEY` option must be set to OFF (the default setting).
- Tables must be referenced by two-part names, *schema.tablename* in the view definition.
- User-defined functions referenced in the view must be created by using the `WITH SCHEMABINDING` option.

- Any user-defined functions referenced in the view must be referenced by two-part names, *<schema>*.*<function>*.
- The data access property of a user-defined function must be `NO SQL`, and external access property must be `NO`.
- Common language runtime (CLR) functions can appear in the select list of the view, but cannot be part of the definition of the clustered index key. CLR functions cannot appear in the WHERE clause of the view or the ON clause of a JOIN operation in the view.
- CLR functions and methods of CLR user-defined types used in the view definition must have the properties set as shown in the following table.

Property	**Note**
DETERMINISTIC = TRUE	Must be declared explicitly as an attribute of the Microsoft .NET Framework method.
PRECISE = TRUE	Must be declared explicitly as an attribute of the .NET Framework method.
DATA ACCESS = NO SQL	Determined by setting DataAccess attribute to DataAccessKind.None and SystemDataAccess attribute to SystemDataAccessKind.None.
EXTERNAL ACCESS = NO	This property defaults to NO for CLR routines.

- The view must be created by using the `WITH SCHEMABINDING` option.
- The view must reference only base tables that are in the same database as the view. The view cannot reference other views.
- If `GROUP BY` is present, the VIEW definition must contain `COUNT_BIG(*)` and must not contain `HAVING`. These `GROUP BY` restrictions are applicable only to the indexed view definition. A query can use an indexed view in its execution plan

even if it does not satisfy these `GROUP BY` restrictions.

- If the view definition contains a `GROUP BY` clause, the key of the unique clustered index can reference only the columns specified in the `GROUP BY` clause.

- The SELECT statement in the view definition must not contain the following Transact-SQL elements:

Transact-SQL elements	**(continued)**	**(continued)**
`COUNT`	ROWSET functions (`OPENDATASOURCE`, `OPENQUERY`, `OPENROWSET`, AND `OPENXML`)	`OUTER` joins (`LEFT`, `RIGHT`, or `FULL`)
Derived table (defined by specifying a `SELECT` statement in the `FROM` clause)	Self-joins	Specifying columns by using `SELECT *` or `SELECT <table_name>.*`
`DISTINCT`	`STDEV`, `STDEVP`, `VAR`, `VARP`, or `AVG`	Common table expression (CTE)
float[1], **text**, **ntext**, **image**, **XML**, or **filestream** columns	Subquery	`OVER` clause, which includes ranking or aggregate window functions
Full-text predicates (`CONTAINS`, `FREETEXT`)	`SUM` function that references a nullable expression	`ORDER BY`
CLR user-defined aggregate function	`TOP`	`CUBE`, `ROLLUP`, or `GROUPING SETS` operators
`MIN`, `MAX`	`UNION`, `EXCEPT`, or `INTERSECT` operators	`TABLESAMPLE`
Table variables	`OUTER APPLY` or `CROSS APPLY`	`PIVOT`, `UNPIVOT`
Sparse column sets	Inline (TVF) or multi-statement table-valued functions (MSTVF)	`OFFSET`

Transact-SQL elements	(continued)	(continued)
`CHECKSUM_AGG`		

[1] The indexed view can contain **float** columns; however, such columns cannot be included in the clustered index key.

Important

Indexed views are not supported on top of temporal queries (queries that use `FOR SYSTEM_TIME` clause).

Recommendations

When you refer to **datetime** and **smalldatetime** string literals in indexed views, we recommend that you explicitly convert the literal to the date type you want by using a deterministic date format style. For a list of the date format styles that are deterministic, see CAST and CONVERT (Transact-SQL). For more information about deterministic and nondeterministic expressions, see the Considerations section in this page.

When you execute DML (such as `UPDATE`, `DELETE` or `INSERT`) on a table referenced by a large number of indexed views, or fewer but very complex indexed views, those indexed views will have to be updated as well during DML execution. As a result, DML query performance may degrade significantly, or in some cases, a query plan cannot even be produced. In such scenarios, test your DML queries before production use, analyze the query plan and tune/simplify the DML statement.

Considerations

The setting of the **large_value_types_out_of_row** option of columns in an indexed view is inherited from the setting of the corresponding column in the base table. This value is set by using sp_tableoption. The default setting for columns formed from expressions is 0. This means that large value types are stored in-row.

Indexed views can be created on a partitioned table, and can themselves be partitioned.

To prevent the Database Engine from using indexed views, include the `OPTION (EXPAND VIEWS)` hint on the query. Also, if any of the listed options are incorrectly set, this will prevent the optimizer from using the indexes on the views. For more information about the `OPTION (EXPAND VIEWS)` hint, see SELECT (Transact-SQL).

All indexes on a view are dropped when the view is dropped. All nonclustered indexes and auto-created statistics on the view are dropped when the clustered index is dropped. User-created statistics on the view are maintained. Nonclustered indexes can be individually dropped. Dropping the clustered index on the view removes the stored result set, and the optimizer returns to processing the view like a standard view.

Indexes on tables and views can be disabled. When a clustered index on a table is disabled, indexes on views associated with the table are also disabled.

Expressions that involve implicit conversion of character strings to **datetime** or **smalldatetime** are considered nondeterministic. For more information, see Nondeterministic conversion of literal date strings into DATE values.

Security

Permissions

Requires **CREATE VIEW** permission in the database and **ALTER** permission on the schema in which the view is being created. If the base table resides within a different schema, the **REFERENCES** permission on the table is required as a minimum.

```
> [!NOTE]
> For the creation of the index on top of the view, the base table must have the same owner as the view. This
is also called ownership-chain. This is usually the case when table and view reside within the same schema,
but it is possible that individual objects have different owners. The column **principal_id** in sys.tables
contains a value if the owner is different from the schema-owner.
```

Using Transact-SQL

To create an indexed view

The following example creates a view and an index on that view. Two queries are included that use the indexed view in the AdventureWorks database.

SQL Copy

```sql
--Set the options to support indexed views.
SET NUMERIC_ROUNDABORT OFF;
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,
   QUOTED_IDENTIFIER, ANSI_NULLS ON;
--Create view with schemabinding.
IF OBJECT_ID ('Sales.vOrders', 'view') IS NOT NULL
   DROP VIEW Sales.vOrders ;
GO
CREATE VIEW Sales.vOrders
   WITH SCHEMABINDING
   AS
      SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Revenue,
         OrderDate, ProductID, COUNT_BIG(*) AS COUNT
      FROM Sales.SalesOrderDetail AS od, Sales.SalesOrderHeader AS o
      WHERE od.SalesOrderID = o.SalesOrderID
      GROUP BY OrderDate, ProductID;
GO
--Create an index on the view.
CREATE UNIQUE CLUSTERED INDEX IDX_V1
   ON Sales.vOrders (OrderDate, ProductID);
GO
--This query can use the indexed view even though the view is
--not specified in the FROM clause.
SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev,
   OrderDate, ProductID
FROM Sales.SalesOrderDetail AS od
```

```
JOIN Sales.SalesOrderHeader AS o
   ON od.SalesOrderID=o.SalesOrderID
      AND ProductID BETWEEN 700 and 800
      AND OrderDate >= CONVERT(datetime,'05/01/2002',101)
   GROUP BY OrderDate, ProductID
   ORDER BY Rev DESC;
GO
--This query can use the above indexed view.
SELECT OrderDate, SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev
FROM Sales.SalesOrderDetail AS od
JOIN Sales.SalesOrderHeader AS o
   ON od.SalesOrderID=o.SalesOrderID
      AND DATEPART(mm,OrderDate)= 3
      AND DATEPART(yy,OrderDate) = 2002
    GROUP BY OrderDate
    ORDER BY OrderDate ASC;
```

For more information, see CREATE VIEW (Transact-SQL).

See Also

- CREATE INDEX
- SET ANSI_NULLS
- SET ANSI_PADDING
- SET ANSI_WARNINGS
- SET ARITHABORT
- SET CONCAT_NULL_YIELDS_NULL
- SET NUMERIC_ROUNDABORT
- SET QUOTED_IDENTIFIER

Is this page helpful?

👍 Yes 👎 No