# Indexing SQL Server relational databases for performance

Indexing in a relational database creates a performance tradeoff that is often overlooked. The more indexes you have, the more likely you will retrieve data from the system quickly. However, it's equally true that the more indexes you have, the longer it will take to insert new data into the system. In this article, you’ll learn about the different kinds of indexes supported by Microsoft SQL Server, the different ways that you can implement indexes, and what you can do to get more read performance than you give up in performance.  
  
Definition of an index  
Indexes are database tools that increase the system's ability to retrieve data by not scanning all of the data records in search of the desired record(s). Indexes can change the organization of data so that it is structured similarly to how it will be accessed for retrieval. The indexes are created on column(s) to help the database find data based upon the value(s) contained in those indexed column(s).  
  
Types of indexes  
There are two indexes supported by Microsoft SQL Server: clustered and nonclustered. Clustered indexes dictate the physical order of data stored in a table. Since a table can only have one physical order, there can only be one clustered index per table. Clustered indexes are efficient when searching for ranges of data since the data is already physically ordered.  
  
Nonclustered indexes do not impact the underlying physical storage, but rather are made up of pointers to data rows. If a clustered index exists, the pointers in nonclustered indexes contain a reference to the location in the clustered index. These indexes are more compact than the data and can be scanned faster than the actual data table.  
  
How to implement indexes  
Some indexes are automatically created by the database. For example, Microsoft SQL Server automatically creates unique indexes to enforce UNIQUE constraints, which ensures no duplicate data can be inserted. Other indexes can be created using the CREATE INDEX statement or through the SQL Server Enterprise Manager, which includes an index creation wizard to guide you.  
  
Getting more performance  
While indexes can provide a performance benefit, they also come at a cost. While SQL Server allows you to create up to 256 nonclustered indexes per table, it is not advisable. Indexes require additional storage space both in memory and on the physical disk drive. They also lead to a decrease in performance when performing insert statements because the data needs to be inserted according to the indexes instead of the first available space, which ensures an insert or update statement will take longer, the more indexes that exist.  
  
Use the following list as a guideline to assist you when creating indexes in Microsoft SQL Server:

* **Choose the right data types:** There are certain data types that are more efficient when used in indexes than others. *Int*, *bigint*, *smallint*, and *tinyint* are all very good data types to index because they have a specific size and are easy to do comparison operations on. Other types such as *char* and *varchar* are much less effective because it is not easy to perform mathematical operations on them and comparison operations take longer.
* **Ensure the indexes are actually used:** When performing queries involving columns that are part of a cluster, it is important to pay attention to how the data is used. When functions are applied to the data columns, it invalidates the advantages of the sort. For example, when a date value is indexed and the date value is converted to a string for comparison purposes, the indexed date value is not used in the query.
* **Pay attention to the order of columns when creating multicolumn indexes:** The indexes are sorted based on the first column and then further sorted according to the order of each additional column in the index. The columns with the less unique data should be listed first in the index to ensure that the data is further sorted as it moves across the index.
* **Limit the number of columns in clustered indexes:** The more columns involved in the clustered index the more data will have to be stored in the nonclustered indexes that contain references to the clustered index. This increases the size of the tables that contain the indexes and thus increases the amount of time to search based on the index.
* **Avoid clustered indexes on frequently updated columns:** Since nonclustered indexes depend upon clustered indexes, if the columns comprising the clustered index are frequently updated, this will cause the row locators stored in the nonclustered indexes to also have to be updated. This leads to increased performance costs for all queries associated with these columns as locking occurs.
* **Split operations (if possible):** When inserts and updates need to be performed frequently on a table, as well as reads, try to separate the tables if possible. All of the inserts and updates can be performed on a table with no indexes, and then later replicated to the other table where there are a heavy number of indexes in place to optimize data reads.
* **Rebuild the indexes properly:** Nonclustered indexes contain pointers to the clustered indexes and thus have a dependency on clustered indexes. When clustered indexes are rebuilt, it can be done by first dropping the index and then using CREATE INDEX to recreate the index, or by including the DROP\_EXISTING clause as a part of the CREATE INDEX statement. Performing the drop and create as separate steps will cause the nonclustered indexes to be rebuilt multiple times rather than just once when the DROP\_EXISTING clause is used.
* **Use fill factorwisely:** Data is stored in contiguous pages that have a set size. When new rows are added to a data page that is full, the system must perform a page split which moves half of the data to a new page. This adds to system overhead and leads to fragmented data. The fill factor allows you to maintain gaps in the data when the index is built. This reduces the number of page splits that occur as data is inserted. The space is maintained only when the index is created and not as data is added or updated. Thus, indexes must be periodically rebuilt in order to continue to utilize the fill factor. The gaps left by the fill factor can lead to slower read performance as more disk accesses are required to read data since it is more spread out; so, it is important to consider whether the number of reads outweigh the number of write operations to determine whether a fill factor other than the default is appropriate for use.

Management decisions  
Better query performance in Microsoft SQL Server can be accomplished through the efficient use of indexes, but the effective use of indexes is dependent on several implementation decisions. Making the right database management decisions regarding the performance tradeoffs of indexing can mean the difference between better performance and bogging down. The guidelines presented in this article will help you make the right decisions for your particular situation.

# Guidelines for Managing Indexes

This section discusses guidelines for managing indexes and contains the following topics:

* [Create Indexes After Inserting Table Data](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006258)
* [Index the Correct Tables and Columns](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006278)
* [Order Index Columns for Performance](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1106306)
* [Limit the Number of Indexes for Each Table](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006312)
* [Drop Indexes That Are No Longer Required](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1106318)
* [Indexes and Deferred Segment Creation](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#CHDCFGBA)
* [Estimate Index Size and Set Storage Parameters](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006354)
* [Specify the Tablespace for Each Index](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006372)
* [Consider Parallelizing Index Creation](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1206377)
* [Consider Creating Indexes with NOLOGGING](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1206400)
* [Understand When to Use Unusable or Invisible Indexes](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#CIHJIDJG)
* [Consider Costs and Benefits of Coalescing or Rebuilding Indexes](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006415)
* [Consider Cost Before Disabling or Dropping Constraints](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#i1006464)

**See Also:**

* + [*Oracle Database Concepts*](http://docs.oracle.com/cd/E11882_01/server.112/e25789/indexiot.htm#CNCPT811) for conceptual information about indexes and indexing, including descriptions of the various indexing schemes offered by Oracle
  + [*Oracle Database Performance Tuning Guide*](http://docs.oracle.com/cd/E11882_01/server.112/e16638/data_acc.htm#PFGRF94788) and [*Oracle Database Data Warehousing Guide*](http://docs.oracle.com/cd/E11882_01/server.112/e25554/indexes.htm#DWHSG006) for information about bitmap indexes
  + [*Oracle Database Data Cartridge Developer's Guide*](http://docs.oracle.com/cd/E11882_01/appdev.112/e10765/dom_idx.htm#ADDCI290) for information about defining domain-specific operators and indexing schemes and integrating them into the Oracle Database server

## Create Indexes After Inserting Table Data

Data is often inserted or loaded into a table using either the SQL\*Loader or an import utility. It is more efficient to create an index for a table after inserting or loading the data. If you create one or more indexes before loading data, the database then must update every index as each row is inserted.

Creating an index on a table that already has data requires sort space. Some sort space comes from memory allocated for the index creator. The amount for each user is determined by the initialization parameter SORT\_AREA\_SIZE. The database also swaps sort information to and from temporary segments that are only allocated during the index creation in the user's temporary tablespace.

Under certain conditions, data can be loaded into a table with SQL\*Loader direct-path load and an index can be created as data is loaded.

**See Also:**

[*Oracle Database Utilities*](http://docs.oracle.com/cd/E11882_01/server.112/e22490/ldr_modes.htm#SUTIL009) for information about using SQL\*Loader for direct-path load

## Index the Correct Tables and Columns

Use the following guidelines for determining when to create an index:

* Create an index if you frequently want to retrieve less than 15% of the rows in a large table. The percentage varies greatly according to the relative speed of a table scan and how the distribution of the row data in relation to the index key. The faster the table scan, the lower the percentage; the more clustered the row data, the higher the percentage.
* To improve performance on joins of multiple tables, index columns used for joins.

**Note:**

Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key.

* Small tables do not require indexes. If a query is taking too long, then the table might have grown from small to large.

**Columns That Are Suitable for Indexing**

Some columns are strong candidates for indexing. Columns with one or more of the following characteristics are candidates for indexing:

* Values are relatively unique in the column.
* There is a wide range of values (good for regular indexes).
* There is a small range of values (good for bitmap indexes).
* The column contains many nulls, but queries often select all rows having a value. In this case, use the following phrase:
* WHERE COL\_X > -9.99 \* power(10,125)

Using the preceding phrase is preferable to:

WHERE COL\_X IS NOT NULL

This is because the first uses an index on COL\_X (assuming that COL\_X is a numeric column).

**Columns That Are Not Suitable for Indexing**

Columns with the following characteristics are less suitable for indexing:

* There are many nulls in the column and you do not search on the not null values.

LONG and LONG RAW columns cannot be indexed.

**Virtual Columns**

You can create unique or non-unique indexes on virtual columns.

## Order Index Columns for Performance

The order of columns in the CREATE INDEX statement can affect query performance. In general, specify the most frequently used columns first.

If you create a single index across columns to speed up queries that access, for example, col1, col2, and col3; then queries that access just col1, or that access just col1 and col2, are also speeded up. But a query that accessed just col2, just col3, or just col2 and col3 does not use the index.

## Limit the Number of Indexes for Each Table

A table can have any number of indexes. However, the more indexes there are, the more overhead is incurred as the table is modified. Specifically, when rows are inserted or deleted, all indexes on the table must be updated as well. Also, when a column is updated, all indexes that contain the column must be updated.

Thus, there is a trade-off between the speed of retrieving data from a table and the speed of updating the table. For example, if a table is primarily read-only, having more indexes can be useful; but if a table is heavily updated, having fewer indexes could be preferable.

## Drop Indexes That Are No Longer Required

Consider dropping an index if:

* It does not speed up queries. The table could be very small, or there could be many rows in the table but very few index entries.
* The queries in your applications do not use the index.
* The index must be dropped before being rebuilt.

**See Also:**

["Monitoring Index Usage"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes004.htm#i1006905)

## Indexes and Deferred Segment Creation

Index segment creation is deferred when the associated table defers segment creation. This is because index segment creation reflects the behavior of the table it is associated with.

**See Also:**

["Understand Deferred Segment Creation"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/tables002.htm#CHDGJAGB) for further information

## Estimate Index Size and Set Storage Parameters

Estimating the size of an index before creating one can facilitate better disk space planning and management. You can use the combined estimated size of indexes, along with estimates for tables, the undo tablespace, and redo log files, to determine the amount of disk space that is required to hold an intended database. From these estimates, you can make correct hardware purchases and other decisions.

Use the estimated size of an individual index to better manage the disk space that the index uses. When an index is created, you can set appropriate storage parameters and improve I/O performance of applications that use the index. For example, assume that you estimate the maximum size of an index before creating it. If you then set the storage parameters when you create the index, fewer extents are allocated for the table data segment, and all of the index data is stored in a relatively contiguous section of disk space. This decreases the time necessary for disk I/O operations involving this index.

The maximum size of a single index entry is approximately one-half the data block size.

Storage parameters of an index segment created for the index used to enforce a primary key or unique key constraint can be set in either of the following ways:

* In the ENABLE ... USING INDEX clause of the CREATE TABLE or ALTER TABLE statement
* In the STORAGE clause of the ALTER INDEX statement

## Specify the Tablespace for Each Index

Indexes can be created in any tablespace. An index can be created in the same or different tablespace as the table it indexes. If you use the same tablespace for a table and its index, it can be more convenient to perform database maintenance (such as tablespace or file backup) or to ensure application availability. All the related data is always online together.

Using different tablespaces (on different disks) for a table and its index produces better performance than storing the table and index in the same tablespace. Disk contention is reduced. But, if you use different tablespaces for a table and its index and one tablespace is offline (containing either data or index), then the statements referencing that table are not guaranteed to work.

## Consider Parallelizing Index Creation

You can parallelize index creation, much the same as you can parallelize table creation. Because multiple processes work together to create the index, the database can create the index more quickly than if a single server process created the index sequentially.

When creating an index in parallel, storage parameters are used separately by each query server process. Therefore, an index created with an INITIAL value of 5M and a parallel degree of 12 consumes at least 60M of storage during index creation.

**See Also:**

[*Oracle Database VLDB and Partitioning Guide*](http://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel.htm#VLDBG010) for information about using parallel execution

## Consider Creating Indexes with NOLOGGING

You can create an index and generate minimal redo log records by specifying NOLOGGING in the CREATE INDEX statement.

**Note:**

Because indexes created using NOLOGGING are not archived, perform a backup after you create the index.

Creating an index with NOLOGGING has the following benefits:

* Space is saved in the redo log files.
* The time it takes to create the index is decreased.
* Performance improves for parallel creation of large indexes.

In general, the relative performance improvement is greater for larger indexes created without LOGGING than for smaller ones. Creating small indexes withoutLOGGING has little effect on the time it takes to create an index. However, for larger indexes the performance improvement can be significant, especially when you are also parallelizing the index creation.

## Understand When to Use Unusable or Invisible Indexes

Use unusable or invisible indexes when you want to improve the performance of bulk loads, test the effects of removing an index before dropping it, or otherwise suspend the use of an index by the optimizer.

**Unusable indexes**

An **unusable index** is ignored by the optimizer and is not maintained by DML. One reason to make an index unusable is to improve bulk load performance. (Bulk loads go more quickly if the database does not need to maintain indexes when inserting rows.) Instead of dropping the index and later re-creating it, which requires you to recall the exact parameters of the CREATE INDEX statement, you can make the index unusable, and then rebuild it.

You can create an index in the unusable state, or you can mark an existing index or index partition unusable. In some cases the database may mark an index unusable, such as when a failure occurs while building the index. When one partition of a partitioned index is marked unusable, the other partitions of the index remain valid.

An unusable index or index partition must be rebuilt, or dropped and re-created, before it can be used. Truncating a table makes an unusable index valid.

Beginning with Oracle Database 11*g* Release 2, when you make an existing index unusable, its index segment is dropped.

The functionality of unusable indexes depends on the setting of the SKIP\_UNUSABLE\_INDEXES initialization parameter. When SKIP\_UNUSABLE\_INDEXES is TRUE(the default), then:

* DML statements against the table proceed, but unusable indexes are not maintained.
* DML statements terminate with an error if there are any unusable indexes that are used to enforce the UNIQUE constraint.
* For nonpartitioned indexes, the optimizer does not consider any unusable indexes when creating an access plan for SELECT statements. The only exception is when an index is explicitly specified with the INDEX() hint.
* For a partitioned index where one or more of the partitions are unusable, the optimizer does not consider the index if it cannot determine at query compilation time if any of the index partitions can be pruned. This is true for both partitioned and nonpartitioned tables. The only exception is when an index is explicitly specified with the INDEX() hint.

When SKIP\_UNUSABLE\_INDEXES is FALSE, then:

* If any unusable indexes or index partitions are present, any DML statements that would cause those indexes or index partitions to be updated are terminated with an error.
* For SELECT statements, if an unusable index or unusable index partition is present but the optimizer does not choose to use it for the access plan, the statement proceeds. However, if the optimizer does choose to use the unusable index or unusable index partition, the statement terminates with an error.

**Invisible Indexes**

Beginning with Oracle Database 11*g* Release 1, you can create invisible indexes or make an existing index invisible. An **invisible index** is ignored by the optimizer unless you explicitly set the OPTIMIZER\_USE\_INVISIBLE\_INDEXES initialization parameter to TRUE at the session or system level. Unlike unusable indexes, an invisible index is maintained during DML statements. Although you can make a partitioned index invisible, you cannot make an individual index partition invisible while leaving the other partitions visible.Using invisible indexes, you can do the following:

* Test the removal of an index before dropping it.
* Use temporary index structures for certain operations or modules of an application without affecting the overall application.

**See Also:**

* ["Creating an Unusable Index"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes003.htm#CIHFIGDG)
* ["Creating an Invisible Index"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes003.htm#BABDHCJD)
* ["Making an Index Unusable"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes004.htm#CIHJCEAJ)
* ["Making an Index Invisible"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes004.htm#BABJJFEE)

## Consider Costs and Benefits of Coalescing or Rebuilding Indexes

Improper sizing or increased growth can produce index fragmentation. To eliminate or reduce fragmentation, you can rebuild or coalesce the index. But before you perform either task weigh the costs and benefits of each option and choose the one that works best for your situation. [Table 21-1](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm#g1007548) is a comparison of the costs and benefits associated with rebuilding and coalescing indexes.

***Table 21-1 Costs and Benefits of Coalescing or Rebuilding Indexes***

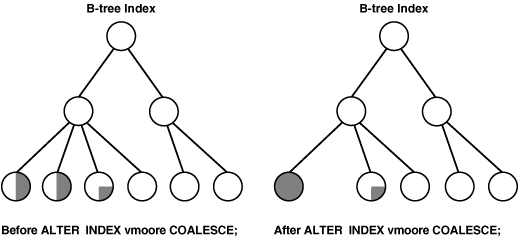
| **Rebuild Index** | **Coalesce Index** |
| --- | --- |
| Quickly moves index to another tablespace | Cannot move index to another tablespace |
| Higher costs: requires more disk space | Lower costs: does not require more disk space |
| Creates new tree, shrinks height if applicable | Coalesces leaf blocks within same branch of tree |
| Enables you to quickly change storage and tablespace parameters without having to drop the original index. | Quickly frees up index leaf blocks for use. |

In situations where you have B-tree index leaf blocks that can be freed up for reuse, you can merge those leaf blocks using the following statement:

ALTER INDEX vmoore COALESCE;

[Figure 21-1](http://docs.oracle.com/cd/E11882_01/server.112/e25494/indexes002.htm" \l "i1006458) illustrates the effect of an ALTER INDEX COALESCE on the index vmoore. Before performing the operation, the first two leaf blocks are 50% full. Therefore, you have an opportunity to reduce fragmentation and completely fill the first block, while freeing up the second.

***Figure 21-1 Coalescing Indexes***

  
[Description of "Figure 21-1 Coalescing Indexes"](http://docs.oracle.com/cd/E11882_01/server.112/e25494/img_text/admin026.htm)

## Consider Cost Before Disabling or Dropping Constraints

Because unique and primary keys have associated indexes, you should factor in the cost of dropping and creating indexes when considering whether to disable or drop a UNIQUE or PRIMARY KEY constraint. If the associated index for a UNIQUE key or PRIMARY KEY constraint is extremely large, you can save time by leaving the constraint enabled rather than dropping and re-creating the large index. You also have the option of explicitly specifying that you want to keep or drop the index when dropping or disabling a UNIQUE or PRIMARY KEY constraint.

**Improving Performance with SQL Server 2008 Indexed Views**

**SQL Server 2008**

32 out of 55 rated this helpful - [Rate this topic](http://msdn.microsoft.com/en-us/library/dd171921(v=sql.100).aspx#feedback)



SQL Server Technical Article

**Writers:** Eric Hanson, Susan Price

**Contributors:** Gail Erickson, Lubor Kollar, Jason Ward

**Technical Reviewer:** Susan Price

**Published:** October 2008

**Applies to:** SQL Server 2008

**Summary:** This document describes the indexed views capability of SQL Server 2005 and SQL Server 2008, including the new support for partition-aligned views added to SQL Server 2008. Indexed views are explained and specific scenarios in which they may provide performance improvements are discussed.

For many years, Microsoft SQL Server has supported the ability to create virtual tables known as *views*. Historically, these views served these main purposes:

* To provide a security mechanism that restricts users to a certain subset of data in one or more base tables.
* To provide a mechanism that allows developers to customize how users can logically view the data stored in base tables.

With SQL Server 2000, the functionality of SQL Server views was expanded to provide system performance benefits. It is possible to create a unique clustered index on a view, as well as nonclustered indexes, to improve data access performance on the most complex queries by precomputing and materializing the view. This is often particularly effective for aggregate views in decision support or data warehouse environments. In SQL Server, a view that has a unique clustered index is referred to as an *indexed view*.

The discussion in this paper applies to SQL Server 2005 and SQL Server 2008. In SQL Server 2005, certain operations on partitioned tables with indexed views required dropping the index and then re-creating the index on the view. In SQL Server 2008, the need to drop an indexed view on a partitioned table during common maintenance operations is greatly reduced, so indexed views are more easily maintained over large partitioned tables.

From the database management system (DBMS) perspective, a view is a description of the data (a form of metadata). When a typical view is created, the metadata is defined by encapsulating a SELECT statement that defines a result set to be represented as a virtual table. When a view is referenced in the FROM clause of another query, this metadata is retrieved from the system catalog and expanded in place of the view's reference. After view expansion, the SQL Server query optimizer compiles a single execution plan for executing the query. The query optimizer searches though a set of possible execution plans for a query, and it chooses the lowest-cost plan it can find, based on estimates of the actual time it will take to execute each query plan.

In the case of a nonindexed view, the portions of the view necessary to solve the query are materialized at run time. Any computations such as joins or aggregations are done during query execution for each query referencing the view [Note1] . After a unique clustered index is created on the view, the view's result set is materialized immediately and persisted in physical storage in the database, saving the overhead of performing this costly operation at execution time.

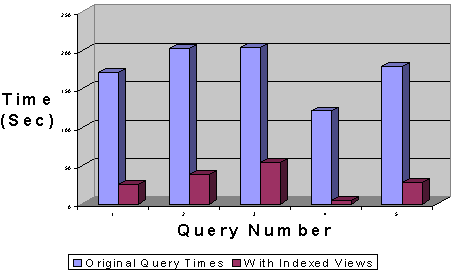
The indexed view can be used in a query execution in two ways. The query can reference the indexed view directly, or, more importantly, the query optimizer can select the view if it determines that the view can be substituted for some or all of the query in the lowest-cost query plan. In the second case, the indexed view is used instead of the underlying tables and their ordinary indexes. The view does not need to be referenced in the query for the query optimizer to use it during query execution. This allows existing applications to benefit from the newly created indexed views without changing those applications.

**Note:** Indexed views are a feature of all versions of SQL Server 2000 and SQL Server 2005. In the Developer and Enterprise editions of SQL Server 2000 and SQL Server 2005, the query processor can use an indexed view to solve queries that structurally match the view, even if they don't refer to the view by name. In other versions, you must reference the view by name and use the NOEXPAND hint on the view reference to query the contents of an indexed view.

Using indexes to improve query performance is not a new concept; however, indexed views provide additional performance benefits that cannot be achieved using standard indexes. Indexed views can increase query performance in the following ways:

* Aggregations can be precomputed and stored in the index to minimize expensive computations during query execution.
* Tables can be prejoined and the resulting data set stored.
* Combinations of joins or aggregations can be stored.

The graph demonstrates the typical performance increases that can be achieved when the query optimizer uses an indexed view. The represented queries varied in complexity (for example, the number of aggregate calculations, the number of tables used, or the number of predicates) and included large multi-million row tables from a real production environment.



**Figure 1** Typical performance improvement with use of an indexed view

Secondary, nonclustered indexes on views can provide additional query performance. Similar to nonclustered indexes on tables, nonclustered indexes on views may provide more options for the query optimizer to choose from during the compilation process. For example, if the query includes columns not covered by the clustered index, the optimizer can choose one or more secondary indexes in the plan and avoid a time-consuming full scan of the indexed view or base tables.

Adding indexes to the schema increases the overhead on the database because the indexes will require ongoing maintenance. Careful consideration should be given to finding the right balance of indexes and maintenance overhead.

Analyze your database workload before implementing indexed views. Use your knowledge of the queries as well as various tools (for example, SQL Server Profiler) to identify the queries that can benefit from indexed views. Frequently occurring aggregations and joins are the best candidates for indexed views. Whether or not a query is asked frequently, it may be a candidate for an indexed view if it takes significant time to answer, and the value of getting the answer quickly is high. For example, some developers find it useful to create indexed views that precompute and store the answers to queries for reports run at the end of each month by senior executives.

Not all queries will benefit from indexed views. Similar to ordinary indexes, if the indexed views are not used, there is no benefit. In this case, not only are performance gains not realized, but the additional cost of disk space, maintenance, and optimization is incurred. However, when indexed views are used, they can provide significant improvements (by orders of magnitude) in data access. This is because the query optimizer uses the precomputed results stored in the indexed view, which substantially reduces the cost of the query execution.

The query optimizer considers indexed views only for queries with nontrivial cost. This avoids situations where trying to match various indexed views during the query optimization costs more than the savings achieved by the indexed view usage. Indexed views are rarely used in queries with a cost of less than 1.

Applications that benefit from the implementation of indexed views include:

* Decision support workloads.
* Data marts.
* Data warehouses.
* Online analytical processing (OLAP) stores and sources.
* Data mining workloads.

From the query type and pattern point of view, the benefiting applications can be characterized as those containing:

* Joins and aggregations of large tables.
* Repeated patterns of queries.
* Repeated aggregations on the same or overlapping sets of columns.
* Repeated joins of the same tables on the same keys.
* Combinations of the above.

On the contrary, online transaction processing (OLTP) systems with many writes, or database applications with frequent updates, may not be able to take advantage of indexed views because of the increased maintenance cost associated with updating both the view and underlying base tables.

In SQL Server Enterprise, the SQL Server query optimizer automatically determines when an indexed view can be used for a given query execution. The view does not need to be referenced directly in the query for the optimizer to use it in the query execution plan. Therefore, existing applications can take advantage of the indexed views without any changes to the application itself; only the indexed views have to be created.

The query optimizer considers several conditions to determine if an indexed view can cover the entire query or a portion of it. These conditions correspond to a single FROM clause in the query and consist of the following:

* The tables in the query FROM clause must be a superset of the tables in the indexed view FROM clause.
* The join conditions in the query must be a superset of the join conditions in the view.
* The aggregate columns in the query must be derivable from a subset of the aggregate columns in the view.
* All expressions in the query select list must be derivable from the view select list or from the tables not included in the view definition.
* One predicate subsumes another if it matches a superset of the rows matched by the other. For example, "T.a=10" subsumes "T.a=10 and T.b=20." Any predicate subsumes itself. The part of the predicate of the view that restricts values of one table must subsume the part of the predicate of the query that restricts the same table. Furthermore, it must do so in a way that SQL Server can verify.

All columns in the query search condition predicates that belong to tables in the view definition must appear in one or more of the following in the view definition:

1. A GROUP BY list.
2. The view select list if there is no GROUP BY.
3. The same or equivalent predicate in the view definition.

Cases (1) and (2) allow SQL Server to apply a query predicate to rows from the view to further restrict the rows of the view. Number (3) is a special case where no filtering is needed on the column, so the column needn't appear in the view.

If the query contains more than one FROM clause (subqueries, derived tables, UNION), the optimizer may select several indexed views to process the query, and apply them to different FROM clauses. [Note2]

Example queries demonstrating these conditions are presented at the end of this document. Allowing the query optimizer to determine which indexes, if any, to use in the query execution plan is the recommended best practice.

**Using the NOEXPAND View Hint**

When SQL Server processes queries that refer to views by name, the definitions of the views normally are expanded until they refer only to base tables. This process is called view expansion. It's a form of macro expansion.

The NOEXPAND view hint forces the query optimizer to treat the view like an ordinary table with a clustered index. It prevents view expansion. The NOEXPAND hint can only be applied if the indexed view is referenced directly in the FROM clause. For example, the following statement includes the indexed view View1 in the FROM clause.

**Transact-SQL**

SELECT Column1, Column2, ... FROM Table1, View1 WITH (NOEXPAND) WHERE ...

Use NOEXPAND if you want to be sure to have SQL Server process a query by reading the view itself instead of reading data from the base tables. If for some reason SQL Server chooses a query plan that processes the query against base tables when you'd prefer that it use the view, consider using NOEXPAND. You must use NOEXPAND in all versions of SQL Server other than Developer and Enterprise editions to have SQL Server process a query against an indexed view directly. You can see a graphical representation of the plan SQL Server chooses for a statement using the SQL Server Management Studio tool Display Estimated Execution Plan feature. Alternatively, you can see different nongraphical representations using SHOWPLAN\_ALL, SHOWPLAN\_TEXT, or SHOWPLAN\_XML. See SQL Server Books Online for a discussion of the different versions of SHOWPLAN.

When processing a query that refers to a view by name, SQL Server always expands the views, unless you add the NOEXPAND hint to the view reference. It attempts to match indexed views to the expanded query, unless you specify the EXPAND VIEWS query hint in an OPTION clause at the end of the query. For example, suppose there is an indexed view View1 in the database. In the following query, View1 is expanded based on its logical definition (its CREATE VIEW statement), and then the EXPAND VIEWS option prevents the indexed view for View1 from being used in the plan to solve the query.

**Transact-SQL**

SELECT Column1, Column2, ... FROM Table1, View1 WHERE ...

OPTION (EXPAND VIEWS)

Use EXPAND VIEWS if you want to be sure to have SQL Server process a query by accessing data directly from the base tables referenced by the query, instead of possibly accessing indexed views. EXPAND views may in some cases help eliminate lock contention that could be experienced with an indexed view. Both NOEXPAND and EXPAND VIEWS can help you evaluate performance with and without use of indexed views when you test your application.

SQL Server 2005 introduced many improvements for indexed views. Starting with SQL Server 2005, the set of indexable views includes those based on:

* Scalar aggregates, including SUM and COUNT\_BIG without GROUP BY.
* Scalar expressions and user-defined functions. For example, given a table T(a int, b int, c int) and a scalar user-defined function dbo.MyUDF(@x int), an indexed view defined on T can contain a computed column such as a+b or dbo.MyUDF(a).
* Persisted imprecise columns. An imprecise column is one whose type is float or real, or a computed column that is derived from a float or real column. In SQL Server 2000, an imprecise column could be used in the select list of an indexed view if it was not part of the index key. An imprecise column could not be used elsewhere inside the view definition either, such as in the WHERE or FROM clauses. SQL Server allows an imprecise column to participate in the key or inside the view definition if the column is persisted in the base table.

Persisted columns include regular columns and computed columns marked PERSISTED. The fundamental reason that imprecise, nonpersisted columns can't participate in indexes or indexed views is that it is necessary to be able to detach a database from one computer and attach it to another. After the move, all computed column values stored in indexes or indexed views must be derivable in exactly the same way on the new hardware as on the old hardware, down to the individual bit. Otherwise, these indexed views are logically corrupted with respect to the new hardware. Because of this corruption, on the new hardware, queries to the indexed views could return different answers depending on whether the plan used the indexed view or the base tables to derive the view data. Furthermore, the indexed views couldn't be maintained correctly on the new computer.

Unfortunately, floating point hardware on different computers (even with the same processor architecture from the same manufacturer) does not always stay 100% the same from version to version of the processor. A firmware upgrade might mean that (a\*b) on the new hardware is not equal to (a\*b) on the old hardware, for some floating point values a and b. For example, the results might be very close, but differ in the least significant bit. Persisting the imprecise computed values before indexing them solves this detach/attach inconsistency problem since all expressions are evaluated on the same computer during database update and maintenance of indexes and indexed views.

* Common Language Runtime (CLR) types. A major new feature of SQL Server 2005 is support for user-defined types (UDTs) and user-defined functions based on the CLR. Indexed views can be defined on CLR UDT columns, or expressions derived from those columns, provided that the columns or expressions are deterministic, and precise, persisted, or both. CLR user-defined aggregates cannot be used in an indexed view.

The capability of the optimizer to match queries to indexed views, and thus use them in query plans, includes:

* New expression types in the SELECT list or condition of a query or view that involve:
  + Scalar expressions, such as (a+b)/2.
  + Scalar aggregates.
  + Scalar user-defined functions.
* Interval subsumption. The optimizer can detect whether interval conditions in an indexed view definition cover, or subsume, interval conditions in a query. For example, the optimizer can determine that "a>10 and a<20" covers "a>12 and a<18."
* Expression equivalence. Certain expressions that can be shown to be the same even though they are syntactically different are treated the same. An example is that "a=b and c<>10" is equivalent to "10<>c and b=a."

In addition, if there is a large number of indexed views in the database, compilation performance for tables on which the views are defined is typically significantly faster in SQL Server 2005 and later than in SQL Server 2000.

Partitioning large fact tables can decrease the time required for maintaining large fact tables. Using indexed views can greatly improve performance of aggregate queries over large fact tables. In SQL Server 2005, an indexed view created on a partitioned table must be dropped before switching a partition in or out of a partitioned table. The indexed view must then be re-created after the partition switch. If the partitioned table is large, re-creating the indexed view may consume considerable time.

SQL Server 2008 includes a new class of indexed views, partition-aligned indexed views, which can dramatically improve the usability of indexed views built on partitioned tables. A partition-aligned indexed view has the same characteristics as a regular indexed view except that the query processor automatically maintains the indexed view when a new partition of the table is switched in. Switching a partition in or out of a partitioned table does not require dropping and re-creating a partition-aligned indexed view.

The availability of partitioned-aligned indexed views substantially improves the manageability of indexed views. As part of your data warehouse load process, a partition of a partition-aligned indexed view can be prepared in a staging area together with a new partition of a partitioned table, and additional maintenance time is not required for re-creating the whole indexed view.

Partition-aligned indexed views are useful whenever you want to do both of the following:

* Partition a table to simplify maintenance.
* Create indexed views over the data in the partitioned tables to improve the performance of queries that access the indexed data.

Partition-aligned indexed views allow table partitioning and indexed views to work together smoothly and efficiently.

For example, suppose you want to store daily sales data, but most of your queries target the most recent day's data. You want to use indexed views to improve performance of data access. You might partition the table by date and load the data for each day into a separate partition. You could switch data in the oldest partition out of the table, either moving the data into an archive table or discarding it, and switch new data into a new partition at the end of each day. You will not need to drop and rebuild your indexed views each day when you switch data in and out of the partitions.

In order for a view to be indexed and automatically maintained during partition switching, the base table and the view must share equivalent partitioning schemes. That means that if the base table and the view use different partition schemes, the two partition schemes must be based on the same, or equivalent, partition functions. For the two partition functions to be equivalent, the partition functions must have identical range clauses (both must specify either LEFT or RIGHT) and the values clauses must specify the same boundary points. More specifically, the view must meet the following requirements to qualify as a partition-aligned indexed view:

* The view definition must qualify for a regular indexed view.
* The partition functions for the base table and the indexed view must define the same number of partitions, must define the same boundary values for the partitions, and must use the same column as the argument to the partition function.
* The projection list of the view definition must include the partitioning key (not a computation of the partitioning key) of the partitioned table.
* If the view definition includes a grouping, the partitioning key must be one of the grouping columns in the view definition.
* If the view refers to multiple tables, exactly one of the tables must be partitioned.

Additional details regarding designing and using partitions and the requirements for using partition-aligned indexed views are documented in SQL Server 2008 Books Online.

An illustration of how to use partition-aligned indexed views appears in the “Examples” section later in this paper.

Identifying an appropriate set of indexes for a database system can be complex. While there are numerous possibilities to consider when designing ordinary indexes, adding indexed views to the schema dramatically increases the complexity of the design and the potential results. For example, indexed views can be used on:

* Any subset of tables referenced in the query.
* Any subset of the conditions in the query for that subset of tables.
* Grouping columns.
* Aggregate functions (for example, SUM).

Indexes on tables and indexed views should be designed concurrently to obtain the best results from each construct. Because both indexes and indexed views may be useful for a given query, designing them separately can lead to redundant recommendations that incur high storage and maintenance overhead. While you tune the physical design of a database, tradeoffs must be made between the performance requirements of a diverse set of queries and updates that the database system must support. Therefore, identifying an appropriate physical design for indexed views is a challenging task, and the Database Tuning Advisor should be used wherever it is possible.

Query optimization cost can increase substantially if there are many indexed views that the query optimizer may consider for a particular query. A query optimizer may consider all indexed views that are defined on any subset of tables in the query. Each view has to be investigated for the potential substitution before it is rejected. This may take some time, especially if there are hundreds of such views for a given query.

A view must meet several requirements before you can create a unique clustered index on it. During the design phase, consider these requirements:

* The view, and all tables referenced in the view, must be in the same database and have the same owner.
* The indexed view does not need to contain all the tables referenced in the query to be used by the optimizer.
* A unique clustered index must be created before any other indexes can be created on the view.
* Certain SET options (discussed later in this document) must be set correctly when the base tables, view, and index are created, and whenever data in the base tables and view are modified. In addition, the query optimizer will not consider the indexed view unless these SET options are correct.
* The view must be created using schema binding, and any user-defined functions referenced in the view must also be created with the SCHEMABINDING option.
* Additional disk space will be required to hold the data defined by the indexed view.

Consider these guidelines when designing indexed views:

* Design indexed views that can be used by several queries or multiple operations.
* For example, an indexed view that contains the SUM of a column and the COUNT\_BIG of a column can be used by queries that contain the functions SUM, COUNT, COUNT\_BIG, or AVG. The queries will be faster because only a small number of rows from the view need to be retrieved rather than the full number of rows from the base tables and a portion of the computations required for performing the AVG function have already been done.
* Keep the index key compact.
* By using the fewest number of columns and bytes possible in the index key, access to the rows of the indexed view can be done more efficiently because the indexed view rows are narrower, and key comparisons are faster than with a wider key. Additionally, the clustered index key is used as a row locator in every nonclustered index defined on the indexed view. The cost of a larger index key increases in proportion to the number of nonclustered indexes on the view.
* Consider the size of the resulting indexed view.
* In the case of pure aggregation, the indexed view may not provide any significant performance gains if its size is similar to the size of the original table.
* Design multiple smaller indexed views that accelerate parts of the process.
* You may not be able to always design an indexed view that addresses the entire query. Should that occur, consider creating several indexed views, each performing a portion of the query.

Consider these examples:

* A frequently executed query aggregates data in one database, aggregates data in another database, and then joins the results. Because an indexed view cannot reference tables from more than one database, you cannot design a single view to perform the entire process. However, you can create an indexed view in each database that does the aggregation for that database. If the optimizer can match the indexed views against existing queries, at least the aggregation processing will be faster, without the need to recode existing queries. Although the join processing is not faster, the overall query is faster because it uses the aggregations stored in the indexed views.
* A frequently executed query aggregates data from several tables, and it then uses UNION to combine the results. UNION is not allowed in an indexed view. You can design views to perform each of the individual aggregation operations. The optimizer can then select the indexed views to speed up queries with no need to recode the queries. While the UNION processing is not improved, the individual aggregation processes are improved.

The Database Engine Tuning Advisor [Note3] is a SQL Server feature that helps database administrators tune their physical database design. Database Engine Tuning Advisor recommends indexed views in addition to recommending indexes on base tables, and it also recommends table and index partitioning strategies. Using Database Engine Tuning Advisor enhances an administrator's ability to determine the combination of indexes, indexed views, and partitioning strategies that optimize the performance of the typical mix of queries executed against a database. Database Engine Tuning Advisor can recommend a wide variety of indexed views. These include ones that take advantage of the new features for indexed views for SQL Server 2005 described in the section “Indexed Views in SQL Server 2005”. Database Engine Tuning Advisor doesn't eliminate the need for good judgment by the database administrator when designing physical storage structures. However, it can simplify the physical database design process. Database Engine Tuning Advisor operates in cooperation with the cost-based query optimizer by proposing a set of hypothetical index, indexed view, and partition structures. Database Engine Tuning Advisor uses the optimizer to estimate the cost of your workload with and without these structures, and it recommends structures that provide low overall cost.

Because the Database Engine Tuning Advisor forces all the required SET options (to ensure the result set is correct), its indexed view creation will succeed. However, your application may not be able to take advantage of the views if its option settings are not set as required. Inserts, updates, or deletes may fail on tables that participate in indexed view definitions if the SET options aren't specified as required.

Indexed views are similarly maintained; however, if the view references several tables, updating any of them may require updating the indexed view. Unlike ordinary indexes, a single row insert into any of the participating tables may cause multiple row changes in an indexed view. This is because the single row may join with multiple rows of another table. The same is true for updates and deletes. Consequently, maintaining an indexed view may be more expensive than maintaining an index on the table. Conversely, maintaining an indexed view with a highly selective condition may be much less expensive than maintaining an index on a table, because most inserts, deletes, and updates to base tables the view references will not affect the view. These operations can be filtered out with respect to the indexed view without accessing other database data.

In SQL Server, some views can be updated. When a view is updatable, the underlying base tables are modified directly through the view using INSERT, UPDATE, and DELETE statements. Creating an index on a view does not prevent the view from being updatable. Updates to an indexed view really cause updates to the base table(s) underlying the view. These updates propagate back to the indexed view automatically as part of indexed view maintenance. For more information about updatable views, see “Modifying Data Through a View” in SQL Server 2008 Books Online.

Consider the following points when you plan indexed views:

* Additional storage is required in the database for the indexed view. The result set of an indexed view is physically persisted in the database in a manner similar to that of typical table storage.
* SQL Server maintains views automatically; therefore, any changes to a base table on which a view is defined may initiate one or more changes in the view indexes. Thus, additional maintenance overhead is incurred.

The net performance improvement achieved by a view is the difference of the total query execution savings offered by the view and the cost to store and maintain the view.

It is relatively easy to approximate the required storage the view will consume. Evaluate the SELECT statement encapsulated by the view definition with the SQL Server Management Studio execution plan option **Display Estimated Execution Plan**. This graphical display will yield an approximation of the number of rows returned by the query and the size of the row. By multiplying these two values together, it is possible to approximate the potential size of the view; however, this is only an approximation. The actual size of the index on the view can be accurately determined only by executing the query in the view definition, or by creating the index on the view.

From the standpoint of automated maintenance considerations performed by SQL Server, the **Display Estimated Execution Plan** option may give some insight on the impact of this overhead. If a statement that modifies the view (UPDATE on the view, INSERT into a base table) is evaluated with SQL Server Management Studio, an execution plan displayed for the statement will include the maintenance operation for that statement. Taking this cost into consideration along with an idea of how many times this operation will occur in the production environment may indicate the potential cost of view maintenance.

As a general recommendation, any modifications or updates to the view or the base tables underlying the view should be performed in batches if possible, rather than singleton operations. This may reduce some overhead in the view maintenance.

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

1. Verify the setting of ANSI\_NULLS is correct for all existing tables that will be referenced in the view.
2. Verify ANSI\_NULLS is set correctly for the current session as shown in the table in “Using SET Options to Obtain Consistent Results” before creating any new tables.
3. Verify ANSI\_NULLS and QUOTED\_IDENTIFIER are set correctly for the current session as shown in the table in “Using SET Options to Obtain Consistent Results” before creating the view.
4. Verify the view definition is deterministic.
5. Create the view using the WITH SCHEMABINDING option.
6. Verify your session's SET options are set correctly as shown in the table in “Using SET Options to Obtain Consistent Results” before creating the unique clustered index on the view.
7. Create the unique clustered index on the view.

Use the OBJECTPROPERTY function to check the value of ANSI\_NULLS and QUOTED\_IDENTIFIER on an existing table or view.

Evaluating the same expression can produce different results in SQL Server if different SET options are enabled for the current session 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. But after CONCAT\_NULL\_YIEDS\_NULL is set to OFF, the same expression produces 'abc'. Indexed views require fixed values for several SET options for the current session and for objects referenced by the view to ensure that the views can be maintained correctly and return consistent results.

The SET options ANSI\_NULLS and QUOTED\_IDENTIFIER of the current session must both be set to ON at the time a view on which you wish to build an index is created. This is because these two options are stored with the view definition in the system catalogs.

The SET options of the current session must be set to the values shown in the Required value column for the current session whenever these operations occur:

* An index is created on a view.
* There is any INSERT, UPDATE, or DELETE operation performed on any table participating in the indexed view.
* The indexed view is used by the query optimizer to produce the query plan.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SET options | Required value | Default server value | .NET SqlClient, OLE DB and ODBC value | DB LIB value |
| ANSI\_NULLS | ON | OFF | ON | OFF |
| ANSI\_PADDING | ON | OFF | ON | OFF |
| ANSI\_WARNINGS | ON | OFF | ON | OFF |
| CONCAT\_NULL\_YIELDS\_NULL | ON | OFF | ON | OFF |
| NUMERIC\_ROUNDABORT | OFF | OFF | OFF | OFF |
| QUOTED\_IDENTIFIER | ON | OFF | ON | OFF |

The following table lists the SET options.

The ARITHABORT option does have to be ON for the current session to create an indexed view, but it is implicitly ON in SQL Server once ANSI\_WARNINGS is ON, so it does not need to be set explicitly. If you are using a .NET SqlClient, OLE DB, or ODBC server connection, you do not have to modify any SET options from their defaults to create, use, and maintain indexed views. All DB LIB values must be set correctly either at the server level using sp\_configure or from the application using the SET command. For more information about SET options, see “Using Options in SQL Server” in SQL Server Books Online.

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, yet the value it returns changes each time it is executed. For more information, see “Deterministic and Nondeterministic Functions” in SQL Server Books Online.

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 in SQL Server when moving a database from one computer to another, such expressions can participate only as non-key columns of indexed views. Deterministic expressions that do not contain float expressions are called precise. Only deterministic expressions that are persisted and/or precise may participate in key columns and WHERE or GROUP BY clauses of indexed views. Persisted expressions are references to stored columns, including regular columns and computed columns marked PERSISTED.

Use the COLUMNPROPERTY function and IsDeterministic property to determine if a view column is deterministic. Use the COLUMNPROPERTY function and IsPrecise property to determine if a deterministic column in a view with SCHEMABINDING is precise. COLUMNPROPERTY returns 1 if the property is TRUE, 0 if FALSE, and NULL for invalid input. For example, in this script the SELECT returns 0 for IsPrecise because the b column is of type real.

**Transact-SQL**

CREATE TABLE T(a int, b real, c as getdate(), d as a+b)

CREATE VIEW VT WITH SCHEMABINDING AS SELECT a, b, c, d FROM dbo.T

SELECT object\_id('VT'), COLUMNPROPERTY(object\_id('VT'),'b','IsPrecise')

You can experiment with COLUMNPROPERTY to find out whether the other columns of T are deterministic and precise.

The set of views that can be indexed is a subset of the set of possible views. Any indexable view can exist with or without an index.

In addition to the requirements listed in the design guidelines, “Using Set Options to Obtain Consistent Results”, and “Using Deterministic Functions” in this white paper, the following requirements must be met before you can create a unique clustered index on a view.

Base tables referenced by the view must have the correct value of the SET option ANSI\_NULLS set at the time the table is created. The OBJECTPROPERTY function can be used to check the value of ANSI\_NULLS on an existing table.

User-defined functions referenced by the view must be created using the WITH SCHEMABINDING option.

The view must meet the following requirements:

* The view must be created using the WITH SCHEMABINDING option.
* Tables must be referenced by the view using two-part names (schemaname.tablename).
* User-defined functions must be referenced by the view using two-part names (schemaname.functionname).
* SET options ANSI\_NULLS and QUOTED\_IDENTIFIER must be set correctly.

To create an index on a view in SQL Server, the view definition must not contain any of the following:

|  |  |
| --- | --- |
| ANY, NOT ANY | OPENROWSET, OPENQUERY, OPENDATASOURCE |
| Arithmetic on imprecise (float, real) values | OPENXML |
| COMPUTE, COMPUTE BY | ORDER BY |
| CONVERT producing an imprecise result | OUTER join |
| COUNT(\*) | References to a base table with a disabled clustered index |
| GROUP BY ALL | References to a table or function in a different database |
| Derived tables (subquery in FROM list) | References to another view |
| DISTINCT | ROWSET functions |
| EXISTS, NOT EXISTS | Self-joins |
| Expressions on aggregate results (for example, SUM(x)+SUM(x)) | STDEV, STDEVP, VAR, VARP, AVG |
| Full-text predicates (CONTAINS, FREETEXT, CONTAINSTABLE, FREETEXTTABLE) | Subqueries |
| Imprecise constants (for example, 2.34e5) | SUM on nullable expressions |
| Inline or table-valued functions | Table hints (for example, NOLOCK) |
| MIN, MAX | text, ntext, image, filestream, or xml columns |
| Nondeterministic expressions | TOP |
| Non-Unicode collations | UNION |
| Contradictions SQL Server can detect that mean the view would be empty (for example, where 0=1 and ...) |  |

**Note:** The indexed view may contain float and real columns; however, such columns cannot be included in the clustered index key if they are nonpersisted computed columns.

If GROUP BY is present, the VIEW definition:

* Must contain COUNT\_BIG(\*).
* Must not contain HAVING, CUBE, ROLLUP, or GROUPING().

These 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.

The index must meet the following requirements:

* The user executing the CREATE INDEX statement must be the view owner.
* 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 index must not be created with the IGNORE\_DUP\_KEY option enabled.

The examples in this section illustrate the use of indexed views with two major groups of queries: aggregations and joins. They also demonstrate the conditions used by the query optimizer when determining if an indexed view is applicable. For information, including a complete list of conditions, see "How the Query Optimizer Uses Indexed Views."

The queries are based on tables in **AdventureWorks**, the sample database provided in SQL Server 2005, and **AdventureWorksDW2008**. For SQL Server 2008, the sample databases **AdventureWorks** and **AdventureWorksDW** are available as separate downloads from CodePlex at<http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=16040>. The queries can be executed as written. You may want to use the **Display Estimated Execution Plan** execution plan option in SQL Server Management Studio to view the plans selected by the query optimizer before and after the views are created. Although the examples demonstrate how the optimizer chooses the lower cost execution plan, the sample databases are too small to show performance gains.

Before you begin working on these examples, make sure your session has the correct options set by running these commands.

**Setup**

**Transact-SQL**

SET ANSI\_NULLS ON

SET ANSI\_PADDING ON

SET ANSI\_WARNINGS ON

SET CONCAT\_NULL\_YIELDS\_NULL ON

SET NUMERIC\_ROUNDABORT OFF

SET QUOTED\_IDENTIFIER ON

SET ARITHABORT ON

The following queries show two methods to return the five products with the largest total discount from the **Sales.SalesOrderDetail** table.

**Query 1**

**Transact-SQL**

SELECT TOP 5 ProductID, Sum(UnitPrice\*OrderQty) -

     Sum(UnitPrice\*OrderQty\*(1.00-UnitPriceDiscount)) AS Rebate

FROM Sales.SalesOrderDetail

GROUP BY ProductID

ORDER BY Rebate DESC

**Query 2**

**Transact-SQL**

SELECT TOP 5 ProductID,

SUM(UnitPrice\*OrderQty\*UnitPriceDiscount) AS Rebate

FROM Sales.SalesOrderDetail

GROUP BY ProductID

ORDER BY Rebate DESC

The execution plan selected by the query optimizer contains:

* A Clustered Index Scan on the Sales.SalesOrderDetail table with a row estimate of 121,317 rows.
* A Hash Match/Aggregate operator that puts the selected rows into a hash table based on the GROUP BY column and computes the SUM aggregation for each row.
* A TOP 5 sort operator based on the ORDER BY clause.

**View 1**

Adding an indexed view that includes the aggregations required for the Rebate column will change the query execution plan for Query 1. On a large table (multi-million rows), the query's performance would also improve significantly.

**Transact-SQL**

CREATE VIEW Vdiscount1 WITH SCHEMABINDING AS

SELECT SUM(UnitPrice\*OrderQty) AS SumPrice,

SUM(UnitPrice\*OrderQty\*(1.00-UnitPriceDiscount)) AS SumDiscountPrice,

COUNT\_BIG(\*) AS Count, ProductID

FROM Sales.SalesOrderDetail

GROUP BY ProductID

GO

CREATE UNIQUE CLUSTERED INDEX VDiscountInd ON Vdiscount1 (ProductID)

The execution plan for the first query shows that the Vdiscount1 view is used by the optimizer. However, the view will not be used by the second query because it does not contain the SUM(UnitPrice\*OrderQty\*UnitPriceDiscount) aggregate. Another indexed view can be created that will address both queries.

**View 2**

**Transact-SQL**

CREATE VIEW Vdiscount2 WITH SCHEMABINDING AS

SELECT SUM(UnitPrice\*OrderQty)AS SumPrice,

SUM(UnitPrice\*OrderQty\*(1.00-UnitPriceDiscount))AS SumDiscountPrice,

SUM(UnitPrice\*OrderQty\*UnitPriceDiscount)AS SumDiscountPrice2,

COUNT\_BIG(\*) AS Count, ProductID

FROM Sales.SalesOrderDetail

GROUP BY ProductID

GO

CREATE UNIQUE CLUSTERED INDEX VDiscountInd ON Vdiscount2 (ProductID)

With this indexed view, after dropping Vdiscount1, the query execution plan for both queries now contains:

* A Clustered Index Scan on the Vdiscount2 view with a row estimate of 266 rows.
* A TOP 5 Sort function based on the ORDER BY clause.

The query optimizer selected the view because it provided the lowest execution cost even though it was not referenced in the query.

**Query 3**

Query 3 is similar to the previous queries, but **ProductID** is replaced by the column **SalesOrderID**, which is not included in the view definition. This violates the condition that all expressions in the query select list on tables in the view definition must be derivable from the view select list in order to use the indexed view in the query plan.

**Transact-SQL**

SELECT TOP 3 SalesOrderID,

SUM(UnitPrice\*OrderQty\*UnitPriceDiscount) OrderRebate

FROM Sales.SalesOrderDetail

GROUP BY SalesOrderID

ORDER BY OrderRebate DESC

A separate indexed view would be required to address this query. Vdiscount2 could be modified to include **SalesOrderID**; however, the resulting view would contain as many rows as the original table and would not provide a performance improvement over using the base table.

**Query 4**

This query produces the average price for each product.

**Transact-SQL**

SELECT p.Name, od.ProductID,

AVG(od.UnitPrice\*(1.00-od.UnitPriceDiscount)) AS AvgPrice,

SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p

WHERE od.ProductID=p.ProductID

GROUP BY p.Name, od.ProductID

Complex aggregates (for example, STDEV, VARIANCE, AVG) cannot be included in the definition of an indexed view. However, indexed views can be used to execute a query containing an AVG by including the simple aggregate functions that, when combined, perform the complex aggregation.

**View3**

This indexed view contains the simple aggregate functions needed to perform an AVG function. When Query 4 is executed after the creation of View3, the execution plan shows the view being used. The optimizer can derive the AVG expression from the view's simple aggregation columns **Price** and **Count**.

**Transact-SQL**

CREATE VIEW View3 WITH SCHEMABINDING AS

SELECT ProductID, SUM(UnitPrice\*(1.00-UnitPriceDiscount)) AS Price,

COUNT\_BIG(\*) AS Count, SUM(OrderQty) AS Units

FROM Sales.SalesOrderDetail

GROUP BY ProductID

GO

CREATE UNIQUE CLUSTERED INDEX iv3 ON View3 (ProductID)

**Query 5**

This query is the same as Query 4, but it includes one additional search condition. View3 will work for this query even though the additional search condition references only columns from a table not included in the view definition.

**Transact-SQL**

SELECT p.Name, od.ProductID,

AVG(od.UnitPrice\*(1.00-UnitPriceDiscount)) AS AvgPrice,

SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p

WHERE od.ProductID=p.ProductID AND p.Name like '%Red%'

GROUP BY p.Name, od.ProductID

**Query 6**

The query optimizer cannot use View3 for this query. The added search condition od.UnitPrice>10 contains a column from the table in the view definition, but the column does not appear in the GROUP BY list, nor does the search predicate appear in the view definition.

**Transact-SQL**

SELECT p.Name, od.ProductID,

AVG(od.UnitPrice\*(1.00-UnitPriceDiscount)) AS AvgPrice,

SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p

WHERE od.ProductID = p.ProductID AND od.UnitPrice > 10

GROUP BY p.Name, od.ProductID

**Query 7**

In contrast, the query optimizer can use View3 for Query 7 because the column defined in the new search condition od.ProductID between 0 and 995 is included in the GROUP BY clause in the view definition.

**Transact-SQL**

SELECT p.Name, od.ProductID,

AVG(od.UnitPrice\*(1.00-UnitPriceDiscount)) AS AvgPrice,

SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p

WHERE od.ProductID = p.ProductID AND od.ProductID between 0 and 995

GROUP BY p.Name, od.ProductID

**View4**

This view will satisfy the conditions for Query 6 by including the columns SumPrice and Count in the view definition to allow the AVG in the query to be computed.

**Transact-SQL**

CREATE VIEW View4 WITH SCHEMABINDING AS

SELECT p.Name, od.ProductID,

SUM(od.UnitPrice\*(1.00-UnitPriceDiscount)) AS SumPrice,

SUM(od.OrderQty) AS Units, COUNT\_BIG(\*) AS Count

FROM Sales.SalesOrderDetail AS od, Production.Product AS p

WHERE od.ProductID = p.ProductID AND od.UnitPrice > 10

GROUP BY p.Name, od.ProductID

GO

CREATE UNIQUE CLUSTERED INDEX VdiscountInd on View4 (Name, ProductID)

**Query 8**

The same index on View4 will also be used for a query where a join to the table Sales.SalesOrderHeader is added. This query meets the condition that the tables listed in the query FROM clause are a superset of the tables in the FROM clause of the indexed view.

**Transact-SQL**

SELECT p.Name, od.ProductID,

AVG(od.UnitPrice\*(1.00-UnitPriceDiscount)) AS AvgPrice,

SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p,

     Sales.SalesOrderHeader AS o

WHERE od.ProductID = p.ProductID AND o.SalesOrderID = od.SalesOrderID

     AND od.UnitPrice > 10

GROUP BY p.Name, od.ProductId

The final two queries are modifications of Query 8. Each modification violates one of the optimizer conditions, and unlike Query 8, they cannot use View4.

**Query 8a**

Query 8a cannot use the indexed view because of the WHERE clause mismatch between **UnitPrice** > 10 in the view definition and **UnitPrice** > 25 in the query, and the fact that**UnitPrice** does not appear in the view. The query search condition predicate must be a superset of the search condition predicates in the view definition.

**Transact-SQL**

SELECT p.Name, od.ProductID, AVG(od.UnitPrice\*(1.00-UnitPriceDiscount))

     AvgPrice, SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p,

     Sales.SalesOrderHeader AS o

WHERE od.ProductID = p.ProductID AND o.SalesOrderID = od.SalesOrderID

     AND od.UnitPrice > 25

GROUP BY p.Name, od.ProductID

**Query 8b**

Observe that the table **Sales.SalesOrderHeader** does not participate in the indexed view View4 definition. In spite of that, adding a predicate on this table will disallow using the indexed view because the added predicate may change or eliminate additional rows participating in the aggregates shown in Query 8b.

**Transact-SQL**

SELECT p.Name, od.ProductID, AVG(od.UnitPrice\*(1.00-UnitPriceDiscount))

     AS AvgPrice, SUM(od.OrderQty) AS Units

FROM Sales.SalesOrderDetail AS od, Production.Product AS p,

     Sales.SalesOrderHeader AS o

WHERE od.ProductID = p.ProductID AND o.SalesOrderID = od.SalesOrderID

     AND od.UnitPrice > 10 AND o.OrderDate > '20040728'

GROUP BY p.Name, od.ProductID

**View 4a**

View4a extends View4 by including the **UnitPrice** column in the SELECT list and GROUP BY clause. Query 8a can use View4a because the **UnitPrice** values that are be greater than 10 can be filtered further to leave only those greater than 25. This is an example of interval subsumption.

**Transact-SQL**

CREATE VIEW View4a WITH SCHEMABINDING AS

SELECT p.Name, od.ProductID, od.UnitPrice,

SUM(od.UnitPrice\*(1.00-UnitPriceDiscount)) AS SumPrice,

SUM(od.OrderQty) AS Units, COUNT\_BIG(\*) AS Count

FROM Sales.SalesOrderDetail AS od, Production.Product AS p

WHERE od.ProductID = p.ProductID AND od.UnitPrice > 10

GROUP BY p.Name, od.ProductID, od.UnitPrice

GO

CREATE UNIQUE CLUSTERED INDEX VdiscountInd

     ON View4a (Name, ProductID, UnitPrice)

**View5**

View5 contains an expression in its select and GROUP BY lists. Notice that **LineTotal** is a computed column, so by itself it is an expression. This expression is in turn nested inside a call to the FLOOR function.

**Transact-SQL**

CREATE VIEW View5 WITH SCHEMABINDING AS

SELECT FLOOR(LineTotal) FloorTotal, COUNT\_BIG(\*) C

FROM Sales.SalesOrderDetail

GROUP BY FLOOR(LineTotal)

GO

CREATE UNIQUE CLUSTERED INDEX iView5 ON View5(FloorTotal)

**Query 9**

Query 9 contains the expression FLOOR(LineTotal) in its select and GROUP BY lists. With the extensions to view matching for expressions added in SQL Server 2005, this query uses the index on View5.

**Transact-SQL**

SELECT TOP 5 FLOOR(LineTotal), Count(\*)

FROM Sales.SalesOrderDetail

GROUP BY FLOOR(LineTotal)

ORDER BY COUNT(\*) DESC

**View6**

View6 stores information about line items for the three days at the end of a month. This clusters together these rows on a small number of pages so that queries to**Sales.SalesOrderDetail** for these days can be satisfied quickly.

**Transact-SQL**

CREATE VIEW View6 WITH SCHEMABINDING AS

SELECT SalesOrderID, SalesOrderDetailID, CarrierTrackingNumber, OrderQty,

     ProductID, SpecialOfferID, UnitPrice, UnitPriceDiscount, rowguid,

     ModifiedDate

FROM Sales.SalesOrderDetail

WHERE ModifiedDate IN ( convert(datetime, '2004-07-31', 120),

                        convert(datetime, '2004-07-30', 120),

                        convert(datetime, '2004-07-29', 120) )

GO

CREATE UNIQUE CLUSTERED INDEX VEndJulyO4Ind

     ON View6(SalesOrderID, SalesOrderDetailID)

GO

**Query 10**

The following query can match View6 and the system can produce a plan that scans the VendJuly04Ind index on the view instead of scanning the entire **Sales.SalesOrderDetail**table. This also demonstrates expression equivalence (because the order of days is different in the query than in the view, and the data formats are different) and predicate subsumption (because the query asks for a subset of the results stored in the view).

**Transact-SQL**

SELECT h.\*, SalesOrderDetailID, CarrierTrackingNumber, OrderQty,

   ProductID, SpecialOfferID, UnitPrice, UnitPriceDiscount, d.rowguid,

   d.ModifiedDate

FROM Sales.SalesOrderHeader as h, Sales.SalesOrderDetail as d

WHERE (d.ModifiedDate = '20040729' OR d.ModifiedDate = '20040730')

and d.SalesOrderID=h.SalesOrderID

**View7**

Developers also sometimes find it convenient to use indexed views to enforce specialized integrity constraints. For example, a constraint you can enforce with an indexed view is "Column a of table T is unique except there may be multiple 0 values in the column." Indexed view View7 enforces this constraint. If you run the following script, it runs successfully until the final insert. That statement is disallowed because it adds a nonzero duplicate value.

**Transact-SQL**

USE tempdb

GO

CREATE TABLE T(a int)

GO

CREATE VIEW View7 WITH SCHEMABINDING

AS SELECT a

FROM dbo.T

WHERE a <> 0

GO

CREATE UNIQUE CLUSTERED INDEX IV on View7(a)

GO

-- legal:

INSERT INTO T VALUES(1)

INSERT INTO T VALUES(2)

INSERT INTO T VALUES(0)

INSERT INTO T VALUES(0) -- duplicate 0

-- disallowed:

INSERT INTO T VALUES(2)

**Query 11**

Queries 11 - 13 are based on a table in the sample database **AdventureWorksDW2008**. The following script creates a partitioned table containing a subset of data in the**AdventureWorksDW2008FactResellerSales** table and then creates a partition-aligned indexed view.

**Transact-SQL**

USE AdventureWorksDW2008;

GO

-- create a partition function

CREATE PARTITION FUNCTION [PF1] (int)

AS RANGE LEFT FOR VALUES (20011231, 20021231, 20031231, 20041231);

GO

-- create the partition scheme

CREATE PARTITION SCHEME [PS1]

AS PARTITION [PF1] ALL to ( [PRIMARY] );

GO

-- create a fact table

CREATE TABLE dbo.FactSales (OrderDateKey INT NOT NULL, ProductKey INT, EmployeeKey INT, SalesAmount MONEY)

ON PS1(OrderDateKey)

INSERT INTO factSales (OrderDateKey, ProductKey, EmployeeKey, SalesAmount)

 SELECT OrderDateKey, ProductKey, EmployeeKey, SalesAmount

 FROM AdventureWorksDW2008.dbo.FactResellerSales

GO

-- create a clustered index - note that it is partitioned using the partition scheme specified

CREATE CLUSTERED INDEX ciFactsales on dbo.factSales (OrderDateKey, ProductKey, EmployeeKey) ON PS1(OrderDateKey)

GO

--create an indexed view

CREATE VIEW dbo.SalesByDateProdEmp  WITH SCHEMABINDING AS

(

SELECT OrderDateKey, ProductKey, EmployeeKey, COUNT\_BIG(\*) AS cnt, SUM(ISNULL(SalesAmount,0)) AS SalesAmount

FROM dbo.FactSales

GROUP BY OrderDateKey, ProductKey, EmployeeKey

)

GO

-- materialize the view

-- uses same partitioning scheme as the partitioned table

CREATE UNIQUE CLUSTERED INDEX uciSalesByDateProdEmp

ON dbo. SalesByDateProdEmp (OrderDateKey, ProductKey, EmployeeKey) ON PS1(OrderDateKey)

GO

**Query 12**

The following script switches the data in PARTITION 1 from the table created in Query 11 into a new archive table.

**Transact-SQL**

-- create an archive table to receive the partition that will be switched out of the partitioned table

-- in this example, the archive table is not partitioned

CREATE TABLE dbo.SalesArchive (OrderDateKey INT NOT NULL, ProductKey INT, EmployeeKey INT, SalesAmount MONEY)

GO

-- create a clustered index to match that on the fact table

CREATE CLUSTERED INDEX ciSalesArchive on dbo.SalesArchive (OrderDateKey, ProductKey, EmployeeKey)

GO

-- create an indexed view with view definition matching SalesByDateProdEmp

-- on table FactSales.

CREATE VIEW dbo.SalesArchiveByDateProdEmp WITH SCHEMABINDING AS

(

SELECT OrderDateKey, ProductKey, EmployeeKey, COUNT\_BIG(\*) AS cnt, SUM(ISNULL(SalesAmount,0)) AS SalesAmount

FROM dbo.SalesArchive

GROUP BY OrderDateKey, ProductKey, EmployeeKey

)

GO

-- materialize the view

-- the indexed view on the nonpartitioned archive table is not partitioned.

CREATE UNIQUE CLUSTERED INDEX uciSalesArchiveByDateProdEmp

ON dbo. SalesArchiveByDateProdEmp (OrderDateKey, ProductKey, EmployeeKey)

GO

-- switch the data in the old partition of the source table [FactSales]

-- and the indexed view [SalesByDateProdEmp]

-- out to the archive table [SalesArchive]

-- and the indexed view [SalesArchiveByDateProdEmp].

ALTER TABLE dbo.FactSales SWITCH PARTITION 1 TO dbo.SalesArchive

GO

**Query 13**

In this script, we switch new data into an empty partition in the table created in Query 11.

**Transact-SQL**

-- create a staging table with data to be switched into the target table

CREATE TABLE dbo.SalesStaging (OrderDateKey INT NOT NULL, ProductKey INT, EmployeeKey INT, SalesAmount MONEY)

ON PS1(OrderDateKey)

GO

-- populate the table

INSERT INTO dbo.SalesStaging VALUES (20050801, 355, 282, 5626.32)

INSERT INTO dbo.SalesStaging VALUES (20050901, 344, 283, 5389.45)

INSERT INTO dbo.SalesStaging VALUES (20050501, 347, 283, 2034.23)

INSERT INTO dbo.SalesStaging VALUES (20050401, 345, 282, 11895.20)

INSERT INTO dbo.SalesStaging VALUES (20050201, 351, 283, 6798.54)

GO

-- create a clustered index on the staging table using the same partitioning scheme

CREATE CLUSTERED INDEX ciSalesStaging

ON SalesStaging(OrderDateKey, ProductKey, EmployeeKey) ON PS1(OrderDateKey)

GO

-- create an indexed view

CREATE VIEW SalesStagingByDateProdEmp WITH SCHEMABINDING AS

(

SELECT OrderDateKey, ProductKey, EmployeeKey, COUNT\_BIG(\*) AS cnt, SUM(ISNULL(SalesAmount,0)) AS SalesAmount

FROM dbo.SalesStaging

GROUP BY OrderDateKey, ProductKey, EmployeeKey

)

GO

-- materialize the view

CREATE UNIQUE CLUSTERED INDEX uciSalesStagingByDateProdEmp

ON SalesStagingByDateProdEmp (OrderDateKey, ProductKey, EmployeeKey) ON PS1(OrderDateKey)

GO

-- switch in the data

ALTER TABLE SalesStaging switch PARTITION 5 TO FactSales PARTITION 5

GO

**Query 14**

This example shows how to use the NOEXPAND hint to rewrite a query to use an indexed view with only a small syntax change compared to the original query referencing the fact table. Suppose you wrote the following query originally.

**Transact-SQL**

SELECT d.FiscalYear, d.FiscalQuarter, p.Color, SUM(f.SalesAmount) AS SalesAmount

FROM dbo.factSales AS f, DimDate d, DimProduct p

WHERE f.OrderDateKey = d.DateKey

AND f.ProductKey = p.ProductKey

AND d.FiscalYear IN (2002, 2003, 2004)

AND p.Color IN ('Red', 'Black', 'Silver')

GROUP BY d.FiscalYear, d.FiscalQuarter, p.Color

ORDER BY d.FiscalYear ASC, d.FiscalQuarter ASC, p.Color ASC;

You might find that indexed view matching didn’t take effect. The following slightly modified query forces use of the index on the view in the query plan, and produces a fast execution reliably.

**Transact-SQL**

SELECT d.FiscalYear, d.FiscalQuarter, p.Color, SUM(f.SalesAmount) AS SalesAmount

FROM dbo.SalesByDateProdEmp AS f WITH(NOEXPAND), DimDate d, DimProduct p

WHERE f.OrderDateKey = d.DateKey

AND f.ProductKey = p.ProductKey

AND d.FiscalYear IN (2002, 2003, 2004)

AND p.Color IN ('Red', 'Black', 'Silver')

GROUP BY d.FiscalYear, d.FiscalQuarter, p.Color

ORDER BY d.FiscalYear ASC, d.FiscalQuarter ASC, p.Color ASC

The book by Adamson identified in the reference list at the end of this paper is a useful reference about how to do manual query rewrites using summary aggregates, similar to the one above, to speed up data warehouse queries. In SQL Server you can use the methods described in the book effectively with indexed views and the NOEXPAND hint.

**Q. Why are there restrictions on the kind of views I can create an index on?**

A. To make sure that it is logically possible to incrementally maintain the view, to restrict the ability to create a view that would be expensive to maintain, and to limit the complexity of the SQL Server implementation. A large set of views is nondeterministic and context-dependent; their contents 'change' independently of DML operations. These can't be indexed. Examples are any views that call GETDATE or SUSER\_SNAME in their definition.

**Q. Why does the first index on a view have to be CLUSTERED and UNIQUE?**

A. It must be UNIQUE to allow easy lookup of records in the view by key value during indexed view maintenance, and to prevent creation of views with duplicates, which would require special logic to maintain. It must be clustered because only a clustered index can enforce uniqueness and store the rows at the same time.

**Q. Why isn't my indexed view being picked up by the query optimizer for use in the query plan?**

A. There are three primary reasons the indexed view may not be being chosen by the optimizer:

* You are using a version other than Enterprise or Developer edition of SQL Server. Only Enterprise and Developer editions support automatic query-to-indexed-view matching. Reference the indexed view by name and include the NOEXPAND hint to have the query processor use the indexed view in all other editions.
* The cost of using the indexed view may exceed the cost of getting the data from the base tables, or the query is so simple that a query against the base tables is fast and easy to find. This often happens when the indexed view is defined on small tables. You can use the NOEXPAND hint if you want to force the query processor to use the indexed view. This may require you to rewrite your query if you don't initially reference the view explicitly. You can get the actual cost of the query with NOEXPAND and compare it to the actual cost of the query plan that doesn't reference the view. If they are close, this may give you confidence that the decision of whether or not to use the indexed view doesn't matter.
* The query optimizer is not matching the query to the indexed view. Double-check the definition of the view and the definition of the query to make sure that a structural match between the two is possible. CASTS, converts, and other expressions that don't logically alter your query result may prevent a match. Also, there are limits to the expression normalization and equivalence and subsumption testing that SQL Server performs. It may not be able to show that some equivalent expressions are the same, or that one expression that is logically subsumed by the other is really subsumed, so it may miss a match.

**Q. I update my data warehouse once a week. Indexed views speed up my queries a lot during the week, but slow down the weekly update. What should I do?**

A. Consider dropping the indexed views before the weekly update, and creating them again afterwards.

**Q. My view has duplicates, but I really want to maintain it. What can I do?**

A. Consider creating a view that groups by all the columns or expressions in the view you want and adds a COUNT\_BIG(\*) column. Then create a unique clustered index on the grouping columns. The grouping process ensures uniqueness. This isn't really the same view, but it might satisfy your needs.

**Q. I have a view defined on top of another view. SQL Server won't let me index the top-level view. What can I do?**

A. Consider expanding the definition of the nested view by hand into the top-level view and then indexing it, indexing the innermost view, or not indexing the view.

**Q. Why do indexed views have to be defined WITH SCHEMABINDING?**

A. So that both of the following conditions are met:

* All objects the view references are unambiguously identified using schemaname.objectname, regardless of which user is accessing the view.
* The objects referred to in the view definition can't be altered in a way that would make the view definition illegal or force SQL Server to re-create the index on the view.

**Q. Why can't I use OUTER JOIN in an indexed view?**

A. Rows can logically disappear from an indexed view based on OUTER JOIN when you insert data into a base table. This makes incrementally updating OUTER JOIN views relatively complex to implement, and the performance of the implementation would be slower than for views based on standard (INNER) JOIN.

Indexed views can substantially improve query performance when used appropriately. An indexed view is persistently stored, meaning that the data can be accessed directly without the need to execute the query that defines the view. This is particularly useful for storing precomputed aggregate data.

Good schema design requires balancing the benefits of indexed views with their costs. Indexed views require additional storage space, and updating the base tables on which a view is defined may require updating the indexed view.

Partition-aligned indexed views extend the usefulness of indexed views to scenarios involving partitioned tables. When large tables are partitioned and data is switched in and out of the partitions, partition-aligned indexed views allow you to maintain the indexed views during switching without incurring additional maintenance cost.

**For more information:**

SQL Server Web site: <http://www.microsoft.com/sqlserver/>

SQL Server TechCenter: <http://technet.microsoft.com/en-us/sqlserver/>

SQL Server DevCenter: <http://msdn.microsoft.com/en-us/sqlserver/>

SQL Server Magazine at [http://www.sqlmag.com](http://www.sqlmag.com/)

*Mastering Data Warehouse Aggregates* by Christopher Adamson, Wiley, 2006

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?

Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20Improving%20Performance%20with%20SQL%20Server%202008%20Indexed%20Views).

**End Notes**

[Note1] The view does not always need to be fully materialized. The query can contain additional predicates, joins, or aggregations that can be applied to the tables and views referenced in the view, which eliminates the need for full materialization.

[Note2] There are exceptional situations when the optimizer may collapse two FROM clauses into one (subquery to join, or derived table to join transformation). If that happens, the indexed view substitution may cover more than one FROM clause in the original query.

[Note3] Database Engine Tuning Advisor is an enhancement of the Index Tuning Wizard found in SQL Server 2000.

Prepared By

Gautam Kumar Gupta

Computer Engineer

Software Department, E-Zone International.

Date: 9/7/12