**General Index Design Guidelines**

**SQL Server 2008 R2**

[Other Versions](javascript:;)

http://i.msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png

* [SQL Server 2008](http://msdn.microsoft.com/en-us/library/ms191195(d=printer,v=sql.100).aspx)
* [SQL Server 2005](http://msdn.microsoft.com/en-us/library/ms191195(d=printer,v=sql.90).aspx)

Experienced database administrators can design a good set of indexes, but this task is very complex, time-consuming, and error-prone even for moderately complex databases and workloads. Understanding the characteristics of your database, queries, and data columns can help you design optimal indexes.

Applies to: SQL Server 2008 R2 and higher versions.

[Database Considerations](javascript:void(0))

When you design an index, consider the following database guidelines:

* Large numbers of indexes on a table affect the performance of INSERT, UPDATE, DELETE, and MERGE statements because all indexes must be adjusted appropriately as data in the table changes.
  + Avoid over-indexing heavily updated tables and keep indexes narrow, that is, with as few columns as possible.
  + Use many indexes to improve query performance on tables with low update requirements, but large volumes of data. Large numbers of indexes can help the performance of queries that do not modify data, such as SELECT statements, because the query optimizer has more indexes to choose from to determine the fastest access method.
* Indexing small tables may not be optimal because it can take the query optimizer longer to traverse the index searching for data than to perform a simple table scan. Therefore, indexes on small tables might never be used, but must still be maintained as data in the table changes.
* Indexes on views can provide significant performance gains when the view contains aggregations, table joins, or a combination of aggregations and joins. The view does not have to be explicitly referenced in the query for the query optimizer to use it. For more information, see [Designing Indexed Views](http://msdn.microsoft.com/en-us/library/ms187864(v=sql.105).aspx).
* Use the Database Engine Tuning Advisor to analyze your database and make index recommendations. For more information, see [Understanding Database Engine Tuning Advisor](http://msdn.microsoft.com/en-us/library/ms188639(v=sql.105).aspx).

[Query Considerations](javascript:void(0))

When you design an index, consider the following query guidelines:

* Create nonclustered indexes on all columns that are frequently used in predicates and join conditions in queries.

|  |
| --- |
| **Important noteImportant** |
| Avoid adding unnecessary columns. Adding too many index columns can adversely affect disk space and index maintenance performance. |

* Covering indexes can improve query performance because all the data needed to meet the requirements of the query exists within the index itself. That is, only the index pages, and not the data pages of the table or clustered index, are required to retrieve the requested data; therefore, reducing overall disk I/O. For example, a query of columns **a** and **b** on a table that has a composite index created on columns **a**, **b**, and **c** can retrieve the specified data from the index alone.
* Write queries that insert or modify as many rows as possible in a single statement, instead of using multiple queries to update the same rows. By using only one statement, optimized index maintenance could be exploited.
* Evaluate the query type and how columns are used in the query. For example, a column used in an exact-match query type would be a good candidate for a nonclustered or clustered index. For more information, see [Query Types and Indexes](http://msdn.microsoft.com/en-us/library/ms189605(v=sql.105).aspx).

[Column Considerations](javascript:void(0))

When you design an index consider the following column guidelines:

* Keep the length of the index key short for clustered indexes. Additionally, clustered indexes benefit from being created on unique or nonnull columns. For more information, see [Clustered Index Design Guidelines](http://msdn.microsoft.com/en-us/library/ms190639(v=sql.105).aspx).
* Columns that are of the ntext, text, image, varchar(max), nvarchar(max), and varbinary(max) data types cannot be specified as index key columns. However, varchar(max), nvarchar(max), varbinary(max), and xml data types can participate in a nonclustered index as nonkey index columns. For more information, see [Index with Included Columns](http://msdn.microsoft.com/en-us/library/ms190806(v=sql.105).aspx).
* An xml data type can only be a key column only in an XML index. For more information, see [Indexes on XML Data Type Columns](http://msdn.microsoft.com/en-us/library/ms191497(v=sql.105).aspx).
* Examine column uniqueness. A unique index instead of a nonunique index on the same combination of columns provides additional information for the query optimizer that makes the index more useful. For more information, see [Unique Index Design Guidelines](http://msdn.microsoft.com/en-us/library/ms187019(v=sql.105).aspx).
* Examine data distribution in the column. Frequently, a long-running query is caused by indexing a column with few unique values, or by performing a join on such a column. This is a fundamental problem with the data and query, and generally cannot be resolved without identifying this situation. For example, a physical telephone directory sorted alphabetically on last name will not expedite locating a person if all people in the city are named Smith or Jones. For more information about data distribution, see [Using Statistics to Improve Query Performance](http://msdn.microsoft.com/en-us/library/ms190397(v=sql.105).aspx).
* Consider using filtered indexes on columns that have well-defined subsets, for example sparse columns, columns with mostly NULL values, columns with categories of values, and columns with distinct ranges of values. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce storage costs. For more information, see [Filtered Index Design Guidelines](http://msdn.microsoft.com/en-us/library/cc280372(v=sql.105).aspx).
* Consider the order of the columns if the index will contain multiple columns. The column that is used in the WHERE clause in an equal to (=), greater than (>), less than (<), or BETWEEN search condition, or participates in a join, should be placed first. Additional columns should be ordered based on their level of distinctness, that is, from the most distinct to the least distinct.

For example, if the index is defined as LastName, FirstName the index will be useful when the search criterion is WHERE LastName = 'Smith' or WHERE LastName = Smith AND FirstName LIKE 'J%'. However, the query optimizer would not use the index for a query that searched only on FirstName (WHERE FirstName = 'Jane').

* Consider indexing computed columns. For more information, see [Creating Indexes on Computed Columns](http://msdn.microsoft.com/en-us/library/ms189292(v=sql.105).aspx).

[Index Characteristics](javascript:void(0))

After you have determined that an index is appropriate for a query, you can select the type of index that best fits your situation. Index characteristics include the following:

* Clustered versus nonclustered
* Unique versus nonunique
* Single column versus multicolumn
* Ascending or descending order on the columns in the index
* Full-table versus filtered for nonclustered indexes

You can also customize the initial storage characteristics of the index to optimize its performance or maintenance by setting an option such as FILLFACTOR. For more information, see [Setting Index Options](http://msdn.microsoft.com/en-us/library/ms188677(v=sql.105).aspx). Also, you can determine the index storage location by using filegroups or partition schemes to optimize performance. For more information, see [Placing Indexes on Filegroups](http://msdn.microsoft.com/en-us/library/ms190433(v=sql.105).aspx).