# INDEXES

## Overview

An index is used to speed up searching in the database. An index can be used to efficiently find all row matching with predicates.

|  |  |
| --- | --- |
| Index type | Description |
| Hash | With a hash index, data is accessed through an in-memory hash table. Hash indexes consume a fixed amount of memory, which is a function of the bucket count. |
| memory-optimized Nonclustered | For memory-optimized nonclustered indexes, memory consumption is a function of the row count and the size of the index key columns |
| Clustered | A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key. The clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values. |
| Nonclustered | A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table. |
| Unique | A unique index ensures that the index key contains no duplicate values and therefore every row in the table or view is in some way unique. Uniqueness can be a property of both clustered and nonclustered indexes. |
| Columnstore | An in-memory columnstore index stores and manages data by using column-based data storage and column-based query processing. Columnstore indexes work well for data warehousing workloads that primarily perform bulk loads and read-only queries. Use the columnstore index to achieve up to 10x query performance gains over traditional row-oriented storage, and up to 7x data compression over the uncompressed data size. |
| Index with included columns | A nonclustered index that is extended to include nonkey columns in addition to the key columns. |
| Index on computed columns | An index on a column that is derived from the value of one or more other columns, or certain deterministic inputs. |
| Filtered | An optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. |
| Spatial | A spatial index provides the ability to perform certain operations more efficiently on spatial objects (*spatial data*) in a column of the **geometry** data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied. |
| XML | A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the **xml** data type column. |
| Full-text | A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data. |

### Heaps (Tables without Clustered Indexes)

A heap is a table without a clustered index. One or more nonclustered indexes can be created on tables stored as a heap. Data is stored in the heap without specifying an order. Usually data is initially stored in the order in which is the rows are inserted into the table, but the Database Engine can move data around in the heap to store the rows efficiently;

**NOTE:** There are sometimes good reasons to leave a table as a heap instead of creating a clustered index, but using heaps effectively is an advanced skill. Most tables should have a carefully chosen clustered index unless a good reason exists for leaving the table as a heap.

#### When to Use a Heap

When a table is stored as a heap, individual rows are identified by reference to a row identifier (RID) consisting of the file number, data page number, and slot on the page. The row id is a small and efficient structure. Sometimes data architects use heaps when data is always accessed through nonclustered indexes and the RID is smaller than a clustered index key.

#### When Not to Use a Heap

* Do not use a heap when the data is frequently returned in a sorted order. A clustered index on the sorting column could avoid the sorting operation.
* Do not use a heap when the data is frequently grouped together. Data must be sorted before it is grouped, and a clustered index on the sorting column could avoid the sorting operation.
* Do not use a heap when ranges of data are frequently queried from the table. A clustered index on the range column will avoid sorting the entire heap.
* Do not use a heap when there are no nonclustered indexes and the table is large. In a heap, all rows of the heap must be read to find any row.

#### Managing Heaps:

* To create a heap, create a table without a clustered index. If a table already has a clustered index, drop the clustered index to return the table to a heap.
* To remove a heap, create a clustered index on the heap.
* To rebuild a heap to reclaim wasted space, create a clustered index on the heap, and then drop that clustered index.

Warning: Creating or dropping clustered indexes requires rewriting the entire table. If the table has nonclustered indexes, all the nonclustered indexes must all be recreated whenever the clustered index is changed. Therefore, changing from a heap to a clustered index structure or back can take a lot of time and require disk space for reordering data in tempdb.

A heap is a table without a clustered index. Heaps have one row in sys.partitions, with index\_id = 0 for each partition used by the heap.

## Clustered and Nonclustered Indexes Described

An index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure (Btree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.

**A table or view can contain the following types of indexes:**

### Clustered

* Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be sorted in only one order.
* The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

#### Create Clustered Indexes

##### **PRIMARY KEY and UNIQUE constraints**

* When you create a PRIMARY KEY constraint, a unique clustered index on the column or columns is automatically. The primary key column cannot allow NULL values.
* When you create a UNIQUE constraint, a unique nonclustered index is created to enforce a UNIQUE constraint by default.
* An index created as part of the constraint is automatically given the same name as the constraint name.

##### **Index independent of a constraint**

You can create a clustered index on a column other than primary key column if a nonclustered primary key constraint was specified.

##### **Limitations and Restrictions**

If a clustered index is created on a heap with several existing nonclustered indexes, all the nonclustered indexes must be rebuilt so that they contain the clustering key value instead of the row identifier (RID). Similarly, if a clustered index is dropped on a table that has several nonclustered indexes, the nonclustered indexes are all rebuilt as part of the DROP operation. This may take significant time on large tables.

### Nonclustered

* Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.
* The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.
* You can add nonkey columns to the leaf level of the nonclustered index(Include columns) to by-pass existing index key limits, and execute fully covered, indexed, queries.
* Both clustered and nonclustered indexes can be unique. This means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value.

Indexes are automatically maintained for a table or view whenever the table data is modified.

Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on table columns.

#### Create Nonclustered Indexes

nonclustered index is an index structure separate from the data stored in a table that reorders one or more selected columns. Nonclustered indexes can often help you find data more quickly than searching the underlying table;

##### **UNIQUE constraints**

When you create a UNIQUE constraint, a unique nonclustered index is created to enforce a UNIQUE

constraint by default. You can specify a unique clustered index if a clustered index on the table does not already exist.

##### **Index independent of a constraint**

By default, a nonclustered index is created if clustered is not specified. The maximum number of

nonclustered indexes that can be created per table is 999. This includes any indexes created by PRIMARY KEY or UNIQUE constraints, but does not include XML indexes.

##### **Nonclustered index on an indexed view**

After a unique clustered index has been created on a view, nonclustered indexes can be created.

### Create Unique Indexes

A unique index guarantees that the index key contains no duplicate values and therefore every row in the table is in some way unique. There are no significant differences between creating a UNIQUE constraint and creating a unique index that is independent of a constraint. Data validation occurs in the same manner, and the query optimizer does not differentiate between a unique index created by a constraint or manually created.

### Create Filtered Indexes

A filtered index is an optimized nonclustered index especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance as well as reduce index maintenance and storage costs compared

with full-table indexes.

**Filtered indexes can provide the following advantages over full-table indexes:**

#### Improved query performance and plan quality

A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table nonclustered index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

#### Reduced index maintenance costs

An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table nonclustered index because it is smaller and is only maintained when the data in the index is changed. It is possible to have a large number of filtered indexes, especially when they contain data that is changed infrequently. Similarly, if a filtered index contains only the frequently modified data, the smaller size of the index reduces the cost of updating the statistics.

#### Reduced index storage costs

Creating a filtered index can reduce disk storage for nonclustered indexes when a full-table index is not necessary. You can replace a full-table nonclustered index with multiple filtered indexes without significantly increasing the storage requirements.

#### Design Considerations

* When a column only has a small number of relevant values for queries, you can create a filtered index on the subset of values. For example, when the values in a column are mostly NULL and the query selects only from the non-NULL values, you can create a filtered index for the non-NULL data rows. The resulting index will be smaller and cost less to maintain than a full-table nonclustered index defined on the same key columns.
* When a table has heterogeneous data rows, you can create a filtered index for one or more categories of data. This can improve the performance of queries on these data rows by narrowing the focus of a query to a specific area of the table. Again, the resulting index will be smaller and cost less to maintain than a full table nonclustered index.

#### Limitations and Restrictions

* You cannot create a filtered index on a view. However, the query optimizer can benefit from a filtered index defined on a table that is referenced in a view.
* **Filtered indexes have the following advantages over indexed views:**
* **Reduced index maintenance costs.** For example, the query processor uses fewer CPU resources to update a filtered index than an indexed view.
* **Improved plan quality.** For example, during query compilation, the query optimizer considers using a filtered index in more situations than the equivalent indexed view.
* **Online index rebuilds.** You can rebuild filtered indexes while they are available for queries. Online index rebuilds are not supported for indexed views
* **Non-unique indexes.** Filtered indexes can be non-unique, whereas indexed views must be unique.
* Filtered indexes are defined on one table and only support simple comparison operators. If you need a filter expression that references multiple tables or has complex logic, you should create a view.
* A column in the filtered index expression does not need to be a key or included column in the filtered index definition if the filtered index expression is equivalent to the query predicate and the query does not return the column in the filtered index expression with the query results.
* A column in the filtered index expression should be a key or included column in the filtered index definition if the query predicate uses the column in a comparison that is not equivalent to the filtered index expression.
* A column in the filtered index expression should be a key or included column in the filtered index definition if the column is in the query result set.
* The clustered index key of the table does not need to be a key or included column in the filtered index definition. The clustered index key is automatically included in all nonclustered indexes, including filtered indexes.
* If the comparison operator specified in the filtered index expression of the filtered index results in an implicit or explicit data conversion, an error will occur if the conversion occurs on the left side of a comparison operator. A solution is to write the filtered index expression with the data conversion operator (CAST or CONVERT) on the right side of the comparison operator.

CREATE NONCLUSTERED INDEX <Index\_name> ON <Table\_Name>(Col1, Col2)

WHERE Col3 IS NOT NULL ;

### Create Indexes with Included Columns

By including nonkey columns, you can create nonclustered indexes that cover more queries. This is because the nonkey columns have the following benefits:

* They can be data types not allowed as index key columns.
* They are not considered by the Database Engine when calculating the number of index key columns or index key size.

An index with nonkey columns can significantly improve query performance when all columns in the query are included in the index either as key or nonkey columns. Performance gains are achieved because the query optimizer can locate all the column values within the index; table or clustered index data is not accessed resulting in fewer disk I/O operations.

**Note:** When an index contains all the columns referenced by a query it is typically referred to as *covering the query*.

#### Design Recommendations

* Redesign nonclustered indexes with a large index key size so that only columns used for searching and lookups are key columns. Make all other columns that cover the query into nonkey columns. In this way, you will have all columns needed to cover the query, but the index key itself is small and efficient.
* Include nonkey columns in a nonclustered index to avoid exceeding the current index size limitations of a maximum of 32 key columns and a maximum index key size of 1,700 bytes (16 key columns and 900 bytes prior to SQL Server 2016). The Database Engine does not consider nonkey columns when calculating the number of index key columns or index key size.

#### Limitations and Restrictions

* Nonkey columns can only be defined on nonclustered indexes.
* All data types except **text**, **ntext**, and **image** can be used as nonkey columns.
* Computed columns that are deterministic and either precise or imprecise can be nonkey columns.
* Computed columns derived from **image**, **ntext**, and **text** data types can be nonkey columns as long as the computed column data type is allowed as a nonkey index column.
* Nonkey columns cannot be dropped from a table unless that table’s index is dropped first.
* Nonkey columns cannot be changed, except to do the following: Change the nullability of the column from NOT NULL to NULL.

## Delete an Index

DROP INDEX <Index\_name> ON <Table\_Name>

Indexes created as the result of a PRIMARY KEY or UNIQUE constraint cannot be deleted by using this method. Instead, the constraint must be deleted. To remove the constraint and corresponding index, use ALTER TABLE with the DROP CONSTRAINT clause in Transact-SQL.

## Modify an Index

ALTER INDEX <Index\_name> ON <Table\_name>

SET ( STATISTICS\_NORECOMPUTE = ON, IGNORE\_DUP\_KEY = ON,

ALLOW\_PAGE\_LOCKS = ON );

Indexes created as the result of a PRIMARY KEY or UNIQUE constraint cannot be modified by using this method. Instead, the constraint must be modified.

## Move an Existing Index to a Different Filegroup

**Limitation and Restrictions**

* If a table has a clustered index, moving the clustered index to a new filegroup moves the table to that filegroup.
* You cannot move indexes created using a UNIQUE or PRIMARY KEY constraint using Management Studio. To move these indexes use the CREATE INDEX statement with the (DROP\_EXISTING=ON) option in Transact- SQL.

CREATE NONCLUSTERED INDEX <Index\_name> ON <Table\_name> (col1,Col2)

WITH (DROP\_EXISTING = ON) ON <FileGroup\_name>;

## Indexes on Computed Columns

You can define indexes on computed columns as long as the following requirements are met:

* Ownership requirements
* Determinism requirements
* Precision requirements
* Data type requirements
* SET option requirements

## SORT\_IN\_TEMPDB Option for Indexes

When you create or rebuild an index, by setting the SORT\_IN\_TEMPDB option to ON you can direct the SQL Server Database Engine to use **tempdb** to store the intermediate sort results that are used to build the index. Although this option increases the amount of temporary disk space that is used to create an index, the option could reduce the time that is required to create or rebuild an index when **tempdb** is on a set of disks different from that of the user database.

When SORT\_IN\_TEMPDB is set to OFF, the default, the sort runs are stored in the destination filegroup.

## Phases of Index Building

**As the Database Engine builds an index, it goes through the following phases:**

* The Database Engine first scans the data pages of the base table to retrieve key values and builds an index leaf row for each data row. When the internal sort buffers have been filled with leaf index entries, the entries are sorted and written to disk as an intermediate sort run. The Database Engine then resumes the data page scan until the sort buffers are again filled. This pattern of scanning multiple data pages followed by sorting and writing a sort run continues until all the rows of the base table have been processed.
* In a clustered index, the leaf rows of the index are the data rows of the table; therefore, the intermediate sort runs contain all the data rows. In a nonclustered index, the leaf rows may contain nonkey columns, but are generally smaller than a clustered index. If the index keys are large, or there are several nonkey columns included in the index, a nonclustered sort run can be large.
* The Database Engine merges the sorted runs of index leaf rows into a single, sorted stream. The sort merge component of the Database Engine starts with the first page of each sort run, finds the lowest key in all the pages, and passes that leaf row to the index create component. The next lowest key is processed, and then the next, and so on. When the last leaf index row is extracted from a sort run page, the process shifts to the next page from that sort run. When all the pages in a sort run extent have been processed, the extent is freed. As each leaf index row is passed to the index create component, it is included in a leaf index page in the buffer. Each leaf page is written as it is filled. As leaf pages are written, the Database Engine also builds the upper levels of the index. Each upper level index page is written when it is filled.

**Note:** If a sort operation is not required or if the sort can be performed in memory, the SORT\_IN\_TEMPDB option is ignored.

## Disable Indexes and Constraints

ALTER INDEX <Index\_Name> ON <Table\_Name> DISABLE;

ALTER INDEX ALL ON <Table\_Name> DISABLE;

Disabling an index prevents user access to the index, and for clustered indexes to the underlying table data. The index definition remains in metadata, and index statistics are kept on nonclustered indexes. Disabling a nonclustered or clustered index on a view physically deletes the index data. Disabling a clustered index on a table prevents access to the data; the data still remains in the table, but is unavailable for data manipulation language (DML) operations until the index is dropped or rebuilt.

## Enable Indexes and Constraints

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD;

CREATE INDEX <Index\_Name> ON <Table\_Name> (Col1,Col2) WITH (DROP\_EXISTING = ON);

DBCC DBREINDEX (<Table\_Name>, <Index\_Name>);

ALTER INDEX ALL ON <Table\_Name> REBUILD;

DBCC DBREINDEX (<Table\_Name>, " ");

## Rename Indexes

Renaming an index replaces the current index name with the new name that you provide. The specified name must be unique within the table or view.

EXEC sp\_rename N'<Old\_Index\_Name>', N’<New\_Index\_Name>’, N’INDEX';

## Set Index Options

* The following options are immediately applied to the index by using the SET clause in the ALTER INDEX statement:
* ALLOW\_PAGE\_LOCKS
* ALLOW\_ROW\_LOCKS
* IGNORE\_DUP\_KEY,
* STATISTICS\_NORECOMPUTE
* The following options can be set when you rebuild an index by using either ALTER INDEX REBUILD or CREATE INDEX WITH DROP\_EXISTING:

PAD\_INDEX, FILLFACTOR, SORT\_IN\_TEMPDB, IGNORE\_DUP\_KEY,

STATISTICS\_NORECOMPUTE, ONLINE, ALLOW\_ROW\_LOCKS, ALLOW\_PAGE\_LOCKS, MAXDOP, DROP\_EXISTING (CREATE INDEX only).

## Disk Space Requirements for Index DDL Operations

Disk space is an important consideration when you create, rebuild, or drop indexes. Inadequate disk space can degrade performance or even cause the index operation to fail.

### Index Operations That Require No Additional Disk Space

The following index operations require no additional disk space:

* ALTER INDEX REORGANIZE; however, log space is required.
* DROP INDEX when you are dropping a nonclustered index.
* DROP INDEX when you are dropping a clustered index offline without specifying the MOVE TO clause and nonclustered indexes do not exist. CREATE TABLE (PRIMARY KEY or UNIQUE constraints)

### Index Operations That Require Additional Disk Space

When a new index structure is created, disk space for both the old (source) and new (target) structures is required in their appropriate files and filegroups.

The following index DDL operations create new index structures and require additional disk space:

* CREATE INDEX
* CREATE INDEX WITH DROP\_EXISTING
* ALTER INDEX REBUILD
* ALTER TABLE ADD CONSTRAINT (PRIMARY KEY or UNIQUE)
* ALTER TABLE DROP CONSTRAINT (PRIMARY KEY or UNIQUE) when the constraint is based on a clustered index
* DROP INDEX MOVE TO (Applies only to clustered indexes.)

### Temporary Disk Space for Sorting

Temporary disk space is required for sorting, unless the query optimizer finds an execution plan that does not require sorting. If the SORT\_IN\_TEMPDB option is set to ON, the largest index must fit into **tempdb**. Although this option increases the amount of temporary disk space that is used to create an index

### Temporary Disk Space for Online Index Operations

When you perform index operations online, additional temporary disk space is required.

If a clustered index is created, rebuilt, or dropped online, a temporary nonclustered index is created to map old bookmarks to new bookmarks. If the SORT\_IN\_TEMPDB option is set to ON, this temporary index is created in **tempdb**. If SORT\_IN\_TEMPDB is set to OFF, the same filegroup or partition scheme as the target index is used.

**Note:** The SORT\_IN\_TEMPDB option cannot be set for DROP INDEX statements. The temporary mapping index is always created in the same filegroup or partition scheme as the target index.

### Transaction Log Disk Space for Index Operations

Large-scale index operations can generate large data loads that can cause the transaction log to fill quickly. To make sure that the index operation can be rolled back, the transaction log cannot be truncated until the index operation has completed; however, the log can be backed up during the index operation. Therefore, the transaction log must have sufficient room to store both the index operation transactions and any concurrent user transactions for the duration of the index operation. This is true for both offline and online index operations.

#### Recomandations:

When you run large-scale index operations, consider the following recommendations:

1. Make sure the transaction log has been backed up and truncated before running large-scale index operations online, and that the log has sufficient space to store the projected index and user transactions.

2. Consider setting the SORT\_IN\_TEMPDB option to ON for the index operation. This separates the index transactions from the concurrent user transactions. The index transactions will be stored in the **tempdb** transaction log, and the concurrent user transactions will be stored in the transaction log of the user database. This allows for the transaction log of the user database to be truncated during the index operation if it is required. Additionally, if the **tempdb** log is not on the same disk as the user database log, the two logs are not competing for the same disk space.

3. Use a database recovery model that allows for minimal logging of the index operation. This may reduce the size of the log and prevent the log from filling the log space.

4. Do not run the online index operation in an explicit transaction. The log will not be truncated until the explicit transaction ends.

**Note:** Verify that the **tempdb** database and transaction log have sufficient disk space to handle the index operation. The **tempdb** transaction log cannot be truncated until the index operation is completed.

### Index Disk Space Example

## Reorganize and Rebuild Indexes

The SQL Server Database Engine automatically modifies indexes whenever

insert, update, or delete operations are made to the underlying data. Over time these modifications can cause the information in the index to become scattered in the database (fragmented). Fragmentation exists when indexes have pages in which the logical ordering, based on the key value, does not match the physical ordering inside the data file. Heavily fragmented indexes can degrade query performance and cause your application to respond slowly, especially scan operations.

Rebuilding an index drops and re-creates the index. This removes fragmentation, reclaims disk space by compacting the pages based on the specified or existing fill factor setting, and reorders the index rows in contiguous pages. When ALL is specified, all indexes on the table are dropped and rebuilt in a single transaction.

Reorganizing an index uses minimal system resources. It defragments the leaf level of clustered and nonclustered indexes on tables and views by physically reordering the leaf-level pages to match the logical, left to right, order of the leaf nodes.

ALTER INDEX <Index\_Name> ON <Table\_Name> REORGANIZE ;

ALTER INDEX ALL ON <Table\_Name> REORGANIZE ;

ALTER INDEX <Indx\_Name> ON <Table\_Name> REBUILD;

ALTER INDEX ALL ON <Table\_Name> REBUILD WITH

( FILLFACTOR = 80, SORT\_IN\_TEMPDB = ON, STATISTICS\_NORECOMPUTE = ON );

## Specify Fill Factor for an Index

The fill-factor option is provided for fine-tuning index data storage and performance. When an index is created or rebuilt, the fill-factor value determines the percentage of space on each leaf-level page to be filled with data, reserving the remainder on each page as free space for future growth. For example, specifying a fill-factor value of 80 means that 20 percent of each leaf-level page will be left empty, providing space for index expansion as data is added to the underlying table. The empty space is reserved between the index rows rather than at the end of the index. The fill-factor value is a percentage from 1 to 100, and the server-wide default is 0 which means that the leaf-level pages are filled to capacity.

**NOTE:** Fill-factor values 0 and 100 are the same in all respects.

## Performance Considerations

### Page Splits

A correctly chosen fill-factor value can reduce potential page splits by providing enough space for index expansion as data is added to the underlying table.When a new row is added to a full index page, the Database Engine moves approximately half the rows to a new page to make room for the new row. This reorganization is known as a page split. A page split makes room for new records, but can take time to perform and is a resource intensive operation.

Also, it can cause fragmentation that causes increased I/O operations. When frequent page splits occur, the index can be rebuilt by using a new or existing fill-factor value to redistribute the data.

### Adding Data to the End of the Table

A nonzero fill factor other than 0 or 100 can be good for performance if the new data is evenly distributed throughout the table. However, if all the data is added to the end of the table, the empty space in the index pages will not be filled. For example, if the index key column is an IDENTITY column, the key for new rows is always increasing and the index rows are logically added to the end of the index. If existing rows will be updated with data that lengthens the size of the rows, use a fill factor of less than 100. The extra bytes on each page will help to minimize page splits caused by extra length in the rows.

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD WITH (FILLFACTOR = 80);

CREATE INDEX <Index\_Name> ON <Table\_Name> (Col1,Col2)

WITH (DROP\_EXISTING = ON, FILLFACTOR = 80);

### Perform Index Operations Online

The ONLINE option allows concurrent user access to the underlying table or clustered index data and any associated nonclustered indexes during these index operations.

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD WITH (ONLINE = ON);

## How Online Index Operations Work

To allow for concurrent user activity during an index data definition language (DDL) operation, the following structures are used during the online index operation: source and pre-existing indexes, target, and for rebuilding a heap or dropping a clustered index online, a temporary mapping index.

### Source and pre-existing indexes

The source is the original table or clustered index data. Pre-existing indexes are any nonclustered indexes that are associated with the source structure. For example, if the online index operation is rebuilding a clustered index that has four associated nonclustered indexes, the source is the existing clustered index and the pre-existing indexes are the nonclustered indexes.

The pre-existing indexes are available to concurrent users for select, insert, update, and delete operations. This includes bulk inserts (supported but not recommended) and implicit updates by triggers and referential integrity constraints. All pre-existing indexes are available for queries and searches. This means they may be selected by the query optimizer and, if necessary, specified in index hints.

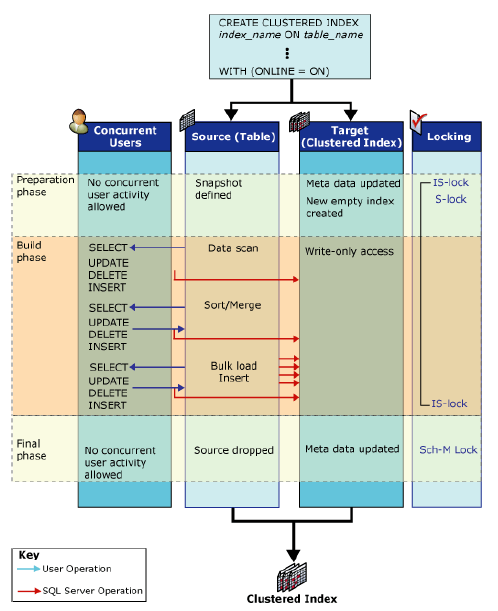
### Target

The target or targets is the new index (or heap) or a set of new indexes that is being created or rebuilt. User insert, update, and delete operations to the source are applied by the SQL Server Database Engine to the target during the index operation. For example, if the online index operation is rebuilding a clustered index, the target is the rebuilt clustered index; the Database Engine does not rebuild nonclustered indexes when a clustered index is rebuilt.

The target index is not searched while processing SELECT statements until the index operation is committed. Internally, the index is marked as write-only.

### Temporary mapping index

Online index operations that create, drop, or rebuild a clustered index also require a temporary mapping index. This temporary index is used by concurrent transactions to determine which records to delete in the new indexes that are being built when rows in the underlying table are updated or deleted. This nonclustered index is created in the same step as the new clustered index (or heap) and does not require a separate sort operation. Concurrent transactions also maintain the temporary mapping index in all their insert, update, and delete operations.



### Guidelines for Online Index Operations

When you perform online index operations, the following guidelines apply:

* Clustered indexes must be created, rebuilt, or dropped offline when the underlying table contains the following large object (LOB) data types: **image**, **ntext**, and **text**.
* Nonunique nonclustered indexes can be created online when the table contains LOB data types but none of these columns are used in the index definition as either key or nonkey (included) columns.
* Indexes on local temp tables cannot be created, rebuilt, or dropped online. This restriction does not apply to indexes on global temp tables.
* Indexes can be resumed from where it stopped after an unexpected failure, database failover, or a **PAUSE** command.

|  |  |  |
| --- | --- | --- |
| **ONLINE INDEX OPERATION** | **EXCLUDED INDEXES** | **OTHER RESTRICTIONS** |
| ALTER INDEX REBUILD | Disabled clustered index or disabled indexed view.  XML index Columnstore index Index on a local temp table | Specifying the keyword ALL may cause the operation to fail when the table contains an excluded index. Additional restrictions on rebuilding disabled indexes apply. |
| CREATE INDEX | XML index Initial unique clustered index on a view Index on a local temp table |  |
| CREATE INDEX WITH DROP\_EXISTING | Disabled clustered index or disabled indexed view Index on a local temp table XML index |  |
| DROP INDEX | Disabled index XML index Nonclustered index Index on a local temp table | Multiple indexes cannot be specified within a single statement. |
| ALTER TABLE ADD CONSTRAINT (PRIMARY KEY or UNIQUE) | Index on a local temp table Clustered index | Only one subclause is allowed at a time. For example, you cannot add and drop PRIMARY KEY or UNIQUE constraints in the same ALTER TABLE statement. |
| ALTER TABLE DROP CONSTRAINT (PRIMARY KEY or UNIQUE) | Clustered index |  |

## Configure Parallel Index Operations

index statements may use multiple processors to perform the scan, sort, and index operations

associated with the index statement just like other queries do. The number of processors used to run a single index statement is determined by the max degree of parallelism configuration option, the current workload, and the index statistics. The max degree of parallelism option determines the maximum number of processors to use in parallel plan execution. If the SQL Server Database Engine detects that the system is busy, the degree of parallelism of the index operation is automatically reduced before statement execution starts.

Parallel index execution and the MAXDOP index option apply to the following Transact-SQL statements:

* CREATE INDEX
* ALTER INDEX REBUILD
* DROP INDEX (This applies to clustered indexes only.)
* ALTER TABLE ADD (index) CONSTRAINT
* ALTER TABLE DROP (clustered index) CONSTRAINT

The MAXDOP index option cannot be specified in the ALTER INDEX REORGANIZE statement.

ALTER INDEX <Index\_Name> ON <Table\_Name> REBUILD WITH (MAXDOP=8);

CREATE INDEX <Index\_Name> ON <Table\_Name> (Col1) WITH (MAXDOP=8);

## Index Properties F1 Help

## Columnstore indexes

*Columnstore indexes* are the standard for storing and querying large data warehousing fact tables. It uses columnbased data storage and query processing to achieve up to **10x query performance** gains in your data warehouse over traditional row-oriented storage, and up to **10x data compression** over the uncompressed data size.

**Columnstore:** A *columnstore* is data that is logically organized as a table with rows and columns, and physically stored in a column-wise data format.

**Rowstore:** A *rowstore* is data that is logically organized as a table with rows and columns, and then physically stored in a rowwise data format. This has been the traditional way to store relational table data.

**Rowgroup:** A *row group* is a group of rows that are compressed into columnstore format at the same time. A rowgroup usually contains the maximum number of rows per rowgroup which is 1,048,576 rows. For high performance and high compression rates, the columnstore index slices the table into groups of rows, called rowgroups, and then compresses each rowgroup in a column-wise manner.

**column segment**

A ***column segment***is a column of data from within the rowgroup

* Each rowgroup contains one column segment for every column in the table.
* Each column segment is compressed together and stored on physical media.

**clustered columnstore index**

A ***clustered columnstore index***is the physical storage for the entire table.

To reduce fragmentation of the column segments and improve performance, the columnstore index might store some data temporarily into a clustered index, which is called a deltastore, and a btree list of IDs for deleted rows.

**deltastore**

Used with clustered column store indexes only, a *deltastore* is a clustered index that improves columnstore compression and performance by storing rows until the number of rows reaches a threshold and are then moved into the columnstore. During a large bulk load, most of the rows go directly to the columnstore without passing through the deltastore.

When the deltastore reaches the maximum number of rows, it becomes closed. A tuple-mover process checks for closed row groups. When it finds the closed rowgroup, it compresses it and stores it into the columnstore.

**nonclustered columnstore index**

A *nonclustered columnstore index* and a clustered columnstore index function the same. The difference is a nonclustered index is a secondary index created on a rowstore table, whereas a clustered columnstore index is the primary storage for the entire table.

**batch execution**

*Batch execution* is a query processing method in which queries process multiple rows together. Queries on columnstore indexes use batch mode execution which improves query performance typically 2-4x. Batch execution is closely integrated with, and optimized around, the columnstore storage format. Batch-mode execution is sometimes known as **vector-based or vectorized execution.**

**Why should I use a columnstore index?**

A columnstore index can provide a very high level of data compression, typically 10x, to reduce your data warehouse storage cost significantly. Plus, for analytics they offer an order of magnitude better performance than a btree index.

**Reasons why columnstore indexes are so fast:**

* Columns store values from the same domain and commonly have similar values, which results in high compression rates. This minimizes or eliminates IO bottleneck in your system while reducing the memory footprint significantly.
* High compression rates improve query performance by using a smaller in-memory footprint. In turn, query performance can improve because SQL Server can perform more query and data operations in-memory.
* Batch execution improves query performance, typically 2-4x, by processing multiple rows together.
* Queries often select only a few columns from a table, which reduces total I/O from the physical media.

**Recommended use cases:**

* Use a clustered columnstore index to store fact tables and large dimension tables for data warehousing workloads. This improves query performance and data compression by up to 10x.
* Use a nonclustered columnstore index to perform analysis in real-time on an OLTP workload.

**How do I choose between a rowstore index and a columnstore index?**

Rowstore indexes perform best on queries that seek into the data, searching for a particular value, or for queries on a small range of values. Use rowstore indexes with transactional workloads since they tend to require mostly table seeks instead of table scans.

Columnstore indexes give high performance gains for analytic queries that scan large amounts of data, especially on large tables. Use columnstore indexes on data warehousing and analytics workloads, especially on fact tables, since they tend to require full table scans rather than table seeks.

**Can I combine rowstore and columnstore on the same table?**

Yes. Beginning with SQL Server 2016, you can create an updatable nonclustered columnstore index on a rowstore table. The columnstore index stores a copy of the chosen columns so you do need extra space for this but it will be compressed on average by 10x. By doing this, you can run analytics on the columnstore index and transactions on the rowstore index at the same time. The column store is updated when data changes in the rowstore table, so both indexes are working against the same data.

Beginning with SQL Server 2016, you can have one or more nonclustered rowstore indexes on a columnstore index. By doing this, you can perform efficient table seeks on the underlying columnstore. Other options become available too. For example, you can enforce a primary key constraint by using a UNIQUE constraint on the rowstore table. Since an non-unique value will fail to insert into the rowstore table, SQL Server cannot insert the value into the columnstore.

Metadata

sys.indexes

sys.index\_columns

sys.partitions sys.internal\_partitions

sys.column\_store\_segments

sys.column\_store\_dictionaries

sys.column\_store\_row\_groups

sys.dm\_db\_column\_store\_row\_group\_operational\_stats

sys.dm\_db\_column\_store\_row\_group\_physical\_stats

sys.dm\_column\_store\_object\_pool

sys.dm\_db\_column\_store\_row\_group\_operational\_stats

sys.dm\_db\_index\_operational\_stats

sys.dm\_db\_index\_physical\_stats