# Index Design Guidelines

**SQL Server Index Design Guide**

Poorly designed indexes and a lack of indexes are primary sources of database application bottlenecks. Designing efficient indexes is paramount to achieving good database and application performance.

**Index Design Basics**

An index is an on-disk or in-memory 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. For on-disk indexes, these keys are stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.

An index stores data logically organized as a table with rows and columns, and physically stored in a row-wise data format called *rowstore* , or stored in a column-wise data format 1 called *columnstore*.

**Index Design Tasks**

The follow tasks make up our recommended strategy for designing indexes:

1. Understand the characteristics of the database itself.

* For example, is it an online transaction processing (OLTP) database with frequent data modifications that must sustain a high throughput. Starting with SQL Server 2014, memory-optimized tables and indexes are especially appropriate for this scenario, by providing a latch-free design.
* Or an example of a Decision Support System (DSS) or data warehousing (OLAP) database that must process very large data sets quickly. Starting with SQL Server 2012, columnstore indexes are especially appropriate for typical data warehousing data sets. Columnstore indexes can transform the data warehousing experience for users by enabling faster performance for common data warehousing queries such as filtering, aggregating, grouping, and star-join queries.

2. Understand the characteristics of the most frequently used queries. For example, knowing that a frequently used query joins two or more tables will help you determine the best type of indexes to use.

3. Understand the characteristics of the columns used in the queries. For example, an index is ideal for columns that have an integer data type and are also unique or nonnull columns. For columns that have well-defined subsets of data, you can use a filtered index in SQL Server 2008 and higher versions.

4. Determine which index options might enhance performance when the index is created or maintained. For example, creating a clustered index on an existing large table would benefit from the ONLINE index option. The ONLINE option allows for concurrent activity on the underlying data to continue while the index is being created or rebuilt.

5. Determine the optimal storage location for the index. A nonclustered index can be stored in the same filegroup as the underlying table, or on a different filegroup. The storage location of indexes can improve query performance by increasing disk I/O performance. For example, storing a nonclustered index on a filegroup that is on a different disk than the table filegroup can improve performance because multiple disks can be read at the same time.

Alternatively, clustered and nonclustered indexes can use a partition scheme across multiple filegroups. Partitioning makes large tables or indexes more manageable by letting you access or manage subsets of data quickly and efficiently, while maintaining the integrity of the overall collection.

**General Index Design Guidelines**

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.

**Database Considerations**

**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.
* Use the Database Engine Tuning Advisor to analyze your database and make index recommendations.

**Query Considerations**

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

* Create nonclustered indexes on the columns that are frequently used in predicates and join conditions in queries. However, you should 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.
* 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

**Column Considerations**

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.
* 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.
* An **xml** data type can only be a key column only in an XML index. This new index can improve querying performance over data stored as XML in SQL Server, allow for much faster indexing of large XML data workloads, and improve scalability by reducing storage costs of the index itself.
* 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.
* 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.
* 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.
* 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.

**Index Characteristics**

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
* Columnstore versus rowstore
* Hash versus nonclustered for Memory-Optimized tables

You can also customize the initial storage characteristics of the index to optimize its performance or

maintenance by setting an option such as FILLFACTOR. Also, you can determine the index storage location by using filegroups or partition schemes to optimize performance.

**Index Placement on Filegroups or Partitions Schemes**

As you develop your index design strategy, you should consider the placement of the indexes on the filegroups associated with the database. Careful selection of the filegroup or partition scheme can improve query performance.

By default, indexes are stored in the same filegroup as the base table on which the index is created. A nonpartitioned clustered index and the base table always reside in the same filegroup. However, you can do the following:

* Create nonclustered indexes on a filegroup other than the filegroup of the base table or clustered index.
* Partition clustered and nonclustered indexes to span multiple filegroups.
* Move a table from one filegroup to another by dropping the clustered index and specifying a new filegroup or partition scheme in the MOVE TO clause of the DROP INDEX statement or by using the CREATE INDEX statement with the DROP\_EXISTING clause.

By creating the nonclustered index on a different filegroup, you can achieve performance gains if the filegroups are using different physical drives with their own controllers. Data and index information can then be read in parallel by the multiple disk heads. For example, if Table\_A on filegroup f1 and Index\_A on filegroup f2 are both being used by the same query, performance gains can be achieved because both filegroups are being fully used without contention. However, if Table\_A is scanned by the query but Index\_A is not referenced, only filegroup f1 is used. This creates no performance gain.

Because you cannot predict what type of access will occur and when it will occur, it could be a better decision to spread your tables and indexes across all filegroups. This would guarantee that all disks are being accessed because all data and indexes are spread evenly across all disks, regardless of which way the data is accessed. This is also a simpler approach for system administrators.

**Partitions across multiple Filegroups**

You can also consider partitioning clustered and nonclustered indexes across multiple filegroups. Partitioned indexes are partitioned horizontally, or by row, based on a partition function.

Partitioning an index can provide the following benefits:

* Provide scalable systems that make large indexes more manageable. OLTP systems, for example, can implement partition-aware applications that deal with large indexes.
* Make queries run faster and more efficiently. When queries access several partitions of an index, the query optimizer can process individual partitions at the same time and exclude partitions that are not affected by the query.

**Index Sort Order Design Guidelines**

When defining indexes, you should consider whether the data for the index key column should be stored in ascending or descending order. Ascending is the default and maintains compatibility with earlier versions of SQL Server. The syntax of the CREATE INDEX, CREATE TABLE, and ALTER TABLE statements supports the keywords ASC (ascending) and DESC (descending) on individual columns in indexes and constraints.

Specifying the order in which key values are stored in an index is useful when queries referencing the table have ORDER BY clauses that specify different directions for the key column or columns in that index. In these cases, the index can remove the need for a SORT operator in the query plan; therefore, this makes the query more efficient.

## Clustered Index Design Guidelines

Clustered indexes sort and store the data rows in the table based on their key values. There can only be one clustered index per table, because the data rows themselves can only be sorted in one order.

* Can be used for frequently used queries.
* Provide a high degree of uniqueness.

NOTE: When you create a PRIMARY KEY constraint, a unique index on the column, or columns, is automatically created. By default, this index is clustered; however, you can specify a nonclustered index when you create the constraint.

* Can be used in range queries.

If the clustered index is not created with the UNIQUE property, the Database Engine automatically adds a 4-byte uniqueifier column to the table. When it is required, the Database Engine automatically adds a uniqueifier value to a row to make each key unique. This column and its values are used internally and cannot be seen or accessed by users.

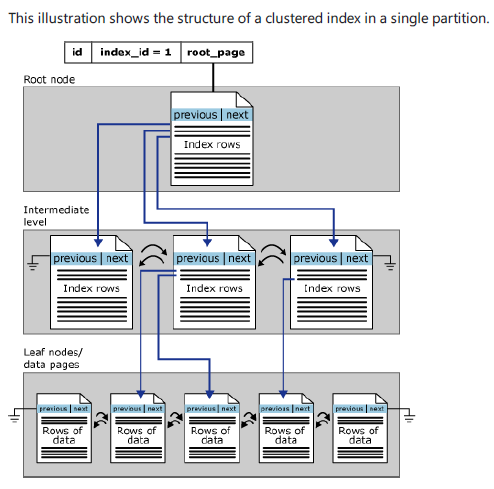
### Clustered Index Architecture

In SQL Server, indexes are organized as B-Trees. Each page in an index B-tree is called an index node. The top node of the B-tree is called the root node. The bottom nodes in the index are called the leaf nodes. Any index levels between the root and the leaf nodes are collectively known as intermediate levels.

In a clustered index, the leaf nodes contain the data pages of the underlying table. The root and intermediate level nodes contain index pages holding index rows. Each index row contains a key value and a pointer to either an intermediate level page in the B-tree, or a data row in the leaf level of the index. The pages in each level of the index are linked in a doubly-linked list.

Clustered indexes have one row in **sys.partitions**, with **index\_id = 1** for each partition used by the index. By default, a clustered index has a single partition. When a clustered index has multiple partitions, each partition has a B-tree structure that contains the data for that specific partition.

Depending on the data types in the clustered index, each clustered index structure will have one or more allocation units in which to store and manage the data for a specific partition. At a minimum, each clustered index will have one IN\_ROW\_DATA allocation unit per partition. The clustered index will also have one LOB\_DATA allocation unit per partition if it contains large object (LOB) columns. It will also have one ROW\_OVERFLOW\_DATA allocation unit per partition if it contains variable length columns that exceed the 8,060 byte row size limit.



### Query Considerations

Before you create clustered indexes, understand how your data will be accessed. Consider using a clustered index for queries that do the following:

* Return a range of values by using operators such as BETWEEN , >, >=, <, and <=.

After the row with the first value is found by using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if a query retrieves records between a range of sales order numbers, a clustered index on the column SalesOrderNumber can quickly locate the row that contains the starting sales order number, and then retrieve all successive rows in the table until the last sales order number is reached.

* Return large result sets.
* Use JOIN clauses; typically these are foreign key columns.
* Use ORDER BY or GROUP BY clauses.

An index on the columns specified in the ORDER BY or GROUP BY clause may remove the need for the Database Engine to sort the data, because the rows are already sorted. This improves query performance.

### Column Considerations

Generally, you should define the clustered index key with as few columns as possible. Consider columns that have one or more of the following attributes:

* Are unique or contain many distinct values

For example, an employee ID uniquely identifies employees. A clustered index or PRIMARY KEY constraint on the EmployeeID column would improve the performance of queries that search for employee information based on the employee ID number.

* Are accessed sequentially
* Defined as IDENTITY .
* Used frequently to sort the data retrieved from a table.

It can be a good idea to cluster, that is physically sort, the table on that column to save the cost of a sort operation every time the column is queried.

**Clustered indexes are not a good choice for the following attributes:**

* Columns that undergo frequent changes

This causes in the whole row to move, because the Database Engine must keep the data values of a row in physical order. This is an important consideration in high-volume transaction processing systems in which data is typically volatile.

* Wide keys

Wide keys are a composite of several columns or several large-size columns. The key values from the clustered index are used by all nonclustered indexes as lookup keys. Any nonclustered indexes defined on the same table will be significantly larger because the nonclustered index entries contain the clustering key and also the key columns defined for that nonclustered index.

## Nonclustered Index Design Guidelines

A nonclustered index contains the index key values and row locators that point to the storage location of the table data. You can create multiple nonclustered indexes on a table or indexed view. Generally, nonclustered indexes should be designed to improve the performance of frequently used queries that are not covered by the clustered index.

Similar to the way you use an index in a book, the query optimizer searches for a data value by searching the nonclustered index to find the location of the data value in the table and then retrieves the data directly from that location. This makes nonclustered indexes the optimal choice for exact match queries because the index contains entries describing the exact location in the table of the data values being searched for in the queries.

### Nonclustered Index Architecture

Nonclustered indexes have the same B-tree structure as clustered indexes, except for the following significant differences:

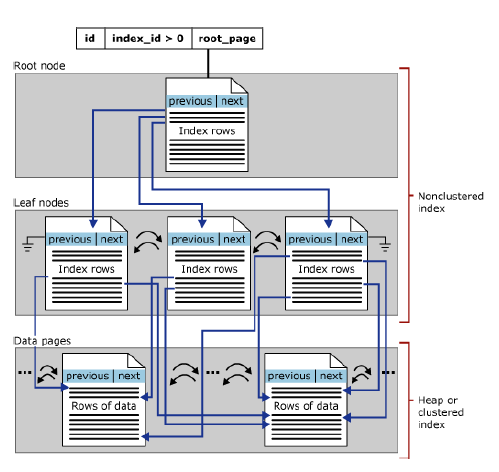
* The data rows of the underlying table are not sorted and stored in order based on their nonclustered keys.
* The leaf layer of a nonclustered index is made up of index pages instead of data pages.

The row locators in nonclustered index rows are either a pointer to a row or are a clustered index key for a row, as described in the following:

* If the table is a heap, which means it does not have a clustered index, the row locator is a pointer to the row. The pointer is built from the file identifier (ID), page number, and number of the row on the page. The whole pointer is known as a Row ID (RID).
* If the table has a clustered index, or the index is on an indexed view, the row locator is the clustered index key for the row.

Nonclustered indexes have one row in sys.partitions with **index\_id** > 1 for each partition used by the index. By default, a nonclustered index has a single partition. When a nonclustered index has multiple partitions, each partition has a B-tree structure that contains the index rows for that specific partition. For example, if a nonclustered index has four partitions, there are four B-tree structures, with one in each partition.

Depending on the data types in the nonclustered index, each nonclustered index structure will have one or more allocation units in which to store and manage the data for a specific partition. At a minimum, each nonclustered index will have one *IN\_ROW\_DATA* allocation unit per partition that stores the index B-tree pages. The nonclustered index will also have one *LOB\_DATA* allocation unit per partition if it contains large object (LOB) columns. Additionally, it will have one *ROW\_OVERFLOW\_DATA* allocation unit per partition if it contains variable length columns that exceed the 8,060 byte row size limit.



### Database Considerations

Consider the characteristics of the database when designing nonclustered indexes.

* Databases or tables with low update requirements, but large volumes of data can benefit from many nonclustered indexes to improve query performance. Consider creating filtered indexes for well-defined subsets of data to improve query performance, reduce index storage costs, and reduce index maintenance costs compared with full-table nonclustered indexes.

Decision Support System applications and databases that contain primarily read-only data can benefit from many nonclustered indexes. The query optimizer has more indexes to choose from to determine the fastest access method, and the low update characteristics of the database mean index maintenance will not impede performance.

* Online Transaction Processing applications and databases that contain heavily updated tables should avoid over-indexing. Additionally, indexes should be narrow, that is, with as few columns as possible.

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.

### Query Considerations

Before you create nonclustered indexes, you should understand how your data will be accessed. Consider using a nonclustered index for queries that have the following attributes:

* Use JOIN or GROUP BY clauses.

Create multiple nonclustered indexes on columns involved in join and grouping operations, and a clustered index on any foreign key columns.

* Queries that do not return large result sets.

Create filtered indexes to cover queries that return a well-defined subset of rows from a large table.

* Contain columns frequently involved in search conditions of a query, such as WHERE clause, that return exact matches.

### Column Considerations

* Cover the query.

Performance gains are achieved when the index contains all columns in the query. 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. Use index with included columns to add covering columns instead of creating a wide index key.

If the table has a clustered index, the column or columns defined in the clustered index are automatically appended to the end of each nonclustered index on the table. This can produce a covered query without specifying the clustered index columns in the definition of the nonclustered index. For example, if a table has a clustered index on column C , a nonclustered index on columns B and A will have as its key values columns B , A , and C .

* Lots of distinct values, such as a combination of last name and first name, if a clustered index is used for other columns.

If there are very few distinct values, such as only 1 and 0, most queries will not use the index because a table scan is generally more efficient. For this type of data, consider creating a filtered index on a distinct value that only occurs in a small number of rows. For example, if most of the values are 0, the query optimizer might use a filtered index for the data rows that contain 1.

### Use Included Columns to Extend Nonclustered Indexes

You can extend the functionality of nonclustered indexes by adding nonkey columns to the leaf level of the nonclustered index. 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 included 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 the query it is typically referred to as covering the query.

While key columns are stored at all levels of the index, nonkey columns are stored only at the leaf level. You can include nonkey columns in a nonclustered index to avoid exceeding the current index size limitations of a maximum of 16 key columns and a maximum index key size of 900 bytes. The Database Engine does not consider nonkey columns when calculating the number of index key columns or index key size.

When you design nonclustered indexes with included columns consider the following guidelines:

* Nonkey columns are defined in the INCLUDE clause of the CREATE INDEX statement.
* Nonkey columns can only be defined on nonclustered indexes on tables or indexed views.
* All data types are allowed except **text**, **ntext**, and **image**.
* Computed columns that are deterministic and either precise or imprecise can be included columns.
* As with key columns, computed columns derived from **image**, **ntext**, and **text** data types can be nonkey (included) columns as long as the computed column data type is allowed as a nonkey index column.
* Column names cannot be specified in both the INCLUDE list and in the key column list.
* Column names cannot be repeated in the INCLUDE list.
* At least one key column must be defined. The maximum number of nonkey columns is 1023 columns. This is the maximum number of table columns minus 1.
* Index key columns, excluding nonkeys, must follow the existing index size restrictions of 16 key columns maximum, and a total index key size of 900 bytes.
* The total size of all nonkey columns is limited only by the size of the columns specified in the INCLUDE clause; for example, **varchar(max)** columns are limited to 2 GB.
* Nonkey columns cannot be dropped from the table unless the 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.
  + Increase the length of **varchar**, **nvarchar**, or **varbinary** columns.

### Performance Considerations

Avoid adding unnecessary columns. Adding too many index columns, key or nonkey, can have the following performance implications:

* Fewer index rows will fit on a page. This could create I/O increases and reduced cache efficiency.
* More disk space will be required to store the index. In particular, adding **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, or **xml** data types as nonkey index columns may significantly increase disk space requirements. This is because the column values are copied into the index leaf level. Therefore, they reside in both the index and the base table.
* Index maintenance may increase the time that it takes to perform modifications, inserts, updates, or deletes, to the underlying table or indexed view.

You will have to determine whether the gains in query performance outweigh the affect to performance during data modification and in additional disk space requirements.

### Unique Index Design Guidelines

A unique index guarantees that the index key contains no duplicate values and therefore every row in the table is in some way unique. Specifying a unique index makes sense only when uniqueness is a characteristic of the data itself.

With multicolumn unique indexes, the index guarantees that each combination of values in the index key is unique. Both clustered and nonclustered indexes can be unique.

**The benefits of unique indexes include the following:**

* Data integrity of the defined columns is ensured.
* Additional information helpful to the query optimizer is provided.
* Creating a PRIMARY KEY or UNIQUE constraint automatically creates a unique index on the specified columns.

**Considerations**

* A unique index, UNIQUE constraint, or PRIMARY KEY constraint cannot be created if duplicate key values exist in the data.
* If the data is unique and you want uniqueness enforced, creating a unique index instead of a nonunique index on the same combination of columns provides additional information for the query optimizer that can produce more efficient execution plans.
* A unique nonclustered index can contain included nonkey columns.

**Filtered Index Design Guidelines**

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, reduce index maintenance costs, and reduce index 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 affected. It is possible to have a large number of filtered indexes, especially when they contain data that is affected infrequently. Similarly, if a filtered index contains only the frequently affected 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.

Filtered indexes are useful when columns contain well-defined subsets of data that queries reference in SELECT statements. Examples are:

* Sparse columns that contain only a few non-NULL values.
* Heterogeneous columns that contain categories of data.
* Columns that contain ranges of values such as dollar amounts, time, and dates.
* Table partitions that are defined by simple comparison logic for column values.

Reduced maintenance costs for filtered indexes are most noticeable when the number of rows in the index is small compared with a full-table index. If the filtered index includes most of the rows in the table, it could cost more to maintain than a full-table index. In this case, you should use a full-table index instead of a filtered index.

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.

**Filtered Indexes for subsets of data**

**Filtered Indexes for heterogeneous data**

Key Columns

Data Conversion Operators in the Filter Predicate

**Hash Index Design Guidelines**

All memory-optimized tables must have at least one index, because it is the indexes that connect the rows together. On a memory-optimized table, every index is also memory-optimized. Hash indexes are one of the possible index types in a memory-optimized table.

**Hash Index Architecture**

A hash index consists of an array of pointers, and each element of the array is called a hash bucket.

* Each bucket is 8 bytes, which are used to store the memory address of a link list of key entries.
* Each entry is a value for an index key, plus the address of its corresponding row in the underlying memoryoptimized table.
* Each entry points to the next entry in a link list of entries, all chained to the current bucket.

**The number of buckets must be specified at index definition time:**

* The lower the ratio of buckets to table rows or to distinct values, the longer the average bucket link list will be.
* Short link lists perform faster than long link lists.
* The maximum number of buckets in hash indexes is 1,073,741,824.

The hash function is applied to the index key columns and the result of the function determines what bucket that key falls into. Each bucket has a pointer to rows whose hashed key values are mapped to that bucket.

**The hashing function used for hash indexes has the following characteristics:**

* SQL Server has one hash function that is used for all hash indexes.
* The hash function is deterministic. The same input key value is always mapped to the same bucket in the hash index.
* Multiple index keys may be mapped to the same hash bucket.
* The hash function is balanced, meaning that the distribution of index key values over hash buckets typically follows a Poisson or bell curve distribution, not a flat linear distribution.
* Poisson distribution is not an even distribution. Index key values are not evenly distributed in the hash buckets.
* If two index keys are mapped to the same hash bucket, there is a *hash collision*. A large number of hash collisions can have a performance impact on read operations. A realistic goal is for 30% of the buckets contain two different key values.

**The performance of a hash index is:**

* Excellent when the predicate in the WHERE clause specifies an **exact** value for each column in the hash index key. A hash index will revert to a scan given an inequality predicate.
* Poor when the predicate in the WHERE clause looks for a **range** of values in the index key.
* Poor when the predicate in the WHERE clause stipulates one specific value for the **first** column of a two column hash index key, but does not specify a value for **other** columns of the key.