**Index**

## 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. For example, if a column is used in several indexes and you execute an UPDATE statement that modifies that column's data, each index that contains that column must be updated as well as the column in the underlying base table (heap or clustered index).
  + 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.

### 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. 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.

### 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.

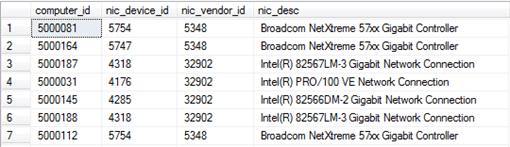
**Why do you need to index your tables?**

Because without an index the SQL server has to scan the entire table to return the requested data. It is like the index page in a book. You check within the index for the keyword you want to learn about. From that point forward, you jump directly to the page where the content belongs, instead of scanning page by page for the material you want to read.

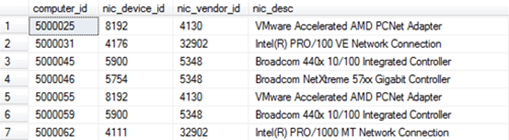
Similarly, a table’s index allows you to locate the data without the need to scan the entire table. You create instead indexes on one or more columns in a table, to aid in the query process. Consider the following example:

SELECT [computer\_id],[nic\_device\_id],[nic\_vendor\_id],[nic\_desc]

FROM [eXpress].[dbo].[nics]



You are about to retrieve all computers with *computer\_id > 5100.* The SQL server will have to scan the entire table in order to return the results, without the presence of an index. An index on the *computer\_id* column will speed up this process, by sorting the column’s values.



Now, if you want to return all data where *computer\_id > 5100*, the SQL server will know how to locate the first value greater than 5100. Why? Because the *computer\_id*column is sorted, eliminating the need to scan the entire table, thus improving performance.

**SQL Index Types**

There are two main index types: Clustered index and Non-Clustered index.

A clustered index alters the way that the rows are physically stored. When you create a clustered index on a column (or a number of columns), the SQL server sorts the table’s rows by that column(s).

It is like a dictionary, where all words are sorted in an alphabetical order. Note, that only one clustered index can be created per table. It alters the way the table is physically stored, it couldn’t be otherwise.

In the example below, all rows are sorted by *computer\_id,* as a clustered index on the computer\_id column has been created.

CREATE CLUSTERED INDEX [IX\_CLUSTERED\_COMPUTER\_ID]

ON [dbo].[nics] ([computer\_id] ASC)

A non-clustered index, on the other hand, does not alter the way the rows are stored in the table. Instead, it creates a completely different object within the table, that contains the column(s) selected for indexing and a pointer back to the table’s rows containing the data.

It is like an index in the last pages of a book. All keywords are sorted and contain a reference back to the appropriate page number. A non-clustered index on the computer\_id column, in the previous example, would look like the table below:

CREATE NONCLUSTERED INDEX [IX\_NONCLUSTERED\_COMPUTER\_ID]

ON [dbo].[nics] ([computer\_id] ASC)

|  |  |
| --- | --- |
| Computer\_id | Row Locator |
| 5000025 | 234 |
| 5000031 | 345 |
| 5000045 | 112 |
| 5000046 | 348 |
| 5000055 | 234 |
| 5000059 | 984 |

Note, that SQL server sorts the indexes efficiently by using a B-tree structure. This is a tree data structure that allows SQL Server to keep data sorted, to allow searches, sequential access, insertions and deletions, in a logarithmic amortized time. This methodology minimizes the number of pages accessed, in order to locate the desired index key, therefore resulting in an improved performance.

Relation between clustered and non-clustered indexes

As explained above, a non-clustered index contains a pointer back to the rowID (RID), of the table, in order to relate the index’s column with the rest of the columns in a row.

But this is not always the case:

If a clustered index already exists in the table, the non-clustered index uses the clustered index’s key as the row locator, instead of the RID reference.

In the example below, when a non-clustered index is created on the *computer\_id* column and a clustered index already exists on the *nic\_desc,* the non-clustered index would look like the table below.

|  |  |
| --- | --- |
| Computer\_id | Row Locator |
| 5000025 | VMware Accelerated AMD PCNet Adapter |
| 5000031 | Intel(R) PRO/100 VE Network Connection |
| 5000045 | Broadcom 440x 10/100 Integrated Controller |
| 5000046 | Broadcom NetXtreme 57xx Gigabit Controller |
| 5000055 | VMware Accelerated AMD PCNet Adapter |
| 5000059 | Broadcom 440x 10/100 Integrated Controller |

Index Benefits and Side Effects

A table without a clustered-index is called a *“heap table”.* A heap table has not its data sorted. The SQL server has to scan the entire table in order to locate the data, in a process called a *“scan”.*

In the case of a clustered index, the data are sorted on the key values (columns) of the index. The SQL server is now able to locate the data by navigating down from the root node, to the branch and finally to the leaf nodes of the B-tree structure of the index. This process called a *“seek”.* The later approach is much faster, when you want to filter or sort the data you want to retrieve.

A non-clustered index, on the other hand, is a completely different object in the table. It contains only a subset of the columns. It also contains a row locator looking back to the table’s rows, or to the clustered index’s key.

Because of its smaller size (subset of columns), a non-clustered index can fit more rows in an index page, therefore resulting to an improved I/O performance. Furthermore a non-clustered index can be allocated to a different FileGroup, which can utilize a different physical storage in order to improve performance even more.

The side effects of indexes are related to the cost of *INSERT*, *UPDATE, MERGE* and *DELETE*statements. Such statements can take longer to execute, in the presence of indexes, as it alters the data in the table, thus to the indexes too.

Imagine the situation of an *INSERT* statement. It has to add new rows in a table with a clustered index. In such case the table rows may need to be re-positioned! Remember…? The clustered index needs to order the data pages themselves! This will cause overhead.

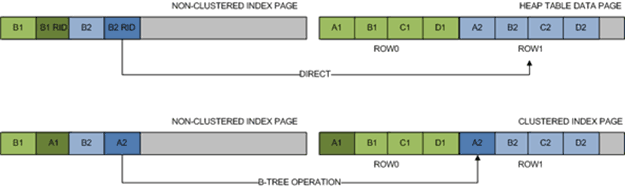
So, it is crucial to take into account the overhead of *INSERT*, *UPDATE* and *DELETE*statements before designing your indexing strategy. Although there is an overhead to the above statements, you have to take into account, that many times, an *UPDATE* or *DELETE*statement will execute in a subset of data. This subset can be defined by a WHERE clause, where indexing may outweigh the additional cost of index updates, because the SQL server will have to find the data before updating them.

As explained above, a non-clustered index includes the clustered index’s key as its row locator, in the presence of a clustered index in the table.

This comes with a cost and a benefit:

The cost has to do with the non-clustered index *bookmark lookup*. What if a query has to return more columns that the ones hosted in the index itself? In the case of a *HEAP* table, the SQL server would have to check the RID of the non-clustered index, in order to navigate directly to the row, where the rest of the columns belong.

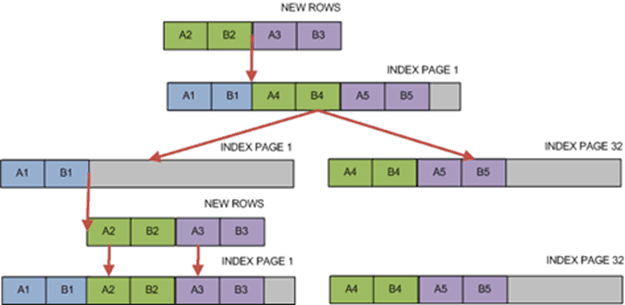
In the case of a clustered index, the SQL server would have to check the row locator of the non-clustered index, in order to do an additional navigation to the B-tree structure of the clustered index, to retrieve the desired row. You see, the row locator does not contain the RID, but the clustered-index key.



On the other hand, there is a benefit. It has to do with the clustered index updates. Imagine the following situation: Two new rows with index key values of A2 and A3 have to be added in the clustered index below.

image_thumb_27

Because this is a clustered index page, its physical structure has to be reallocated in order to fit A2 and A3 between A1 and A4. It has to maintain index’s order. Since there is no free space in the index page to accommodate these changes,  a *page split* will occur. Now, there is enough space to fit A2 and A3 between A1 and A4.



The goal achieved and the order maintained within the index. But imagine what would have happened if the non-clustered index was looking at the RID, instead of the clustered index’s key? It would have to change its row locators to reflect the changes. This could have been a huge performance hit! Especially, in the case of large clustered indexes.

Instead of the RID, the row locators now point at the clustered index key. Meaning, that there is no longer needed to change its values. This is quite a benefit if you think of the large clustered indexes, that are usually maintained in many tables.

Clustered indexes VS non-clustered indexes

|  |  |  |
| --- | --- | --- |
|  | CLUSTERED | NON-CLUSTERED |
| PROS | * Fast to return large range of data * Fast for presorted results | * Wide keys do not reflect on other indexes * Frequently updated key columns do not reflect on other indexes * Can be assigned on different FileGroup * Many non-clustered indexes per table * Smaller size than clustered indexes due to column subsets |
| CONS | * Frequently updated key columns reflect on non-clustered indexes * Wide keys increase the size of the non-clustered indexes * Only one clustered index per table | * Generally slower than clustered indexes due to bookmark lookup (except for covering indexes). * Not recommended for returning large data sets (except for covering indexes). |

## Columnstore Index Design Guidelines

### olumnstore Index Architecture

Knowing these basics will make it easier to understand other columnstore articles that explain how to use them effectively.

#### Data storage uses columnstore and rowstore compression

When discussing columnstore indexes, we use the terms rowstore and columnstore to emphasize the format for the data storage. Columnstore indexes use both types of storage.

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

A columnstore index physically stores most of the data in columnstore format. In columnstore format, the data is compressed and uncompressed as columns. There is no need to uncompress other values in each row that are not requested by the query. This makes it fast to scan an entire column of a large table.

* A **rowstore** is data that is logically organized as a table with rows and columns, and then physically stored in a row-wise data format. This has been the traditional way to store relational table data such as a heap or clustered B-tree index.

A columnstore index also physically stores some rows in a rowstore format called a deltastore. The deltastore,also called delta rowgroups, is a holding place for rows that are too few in number to qualify for compression into the columnstore. Each delta rowgroup is implemented as a clustered B-tree index.

* The **deltastore** is a holding place for rows that are too few in number to be compressed into the columnstore. The deltastore stores the rows in rowstore format.

## Indexes and Constraints

Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on table columns. For example, when you create a table with a UNIQUE constraint, Database Engine automatically creates a non-clustered index. If you configure a PRIMARY KEY, Database Engine automatically creates a clustered index, unless a clustered index already exists. When you try to enforce a PRIMARY KEY constraint on an existing table and a clustered index already exists on that table, SQL Server enforces the primary key using a nonclustered index.

## How Indexes are used by the Query Optimizer

Well-designed indexes can reduce disk I/O operations and consume fewer system resources therefore improving query performance. Indexes can be helpful for a variety of queries that contain SELECT, UPDATE, DELETE, or MERGE statements. Consider the query SELECT Title, HireDate FROM HumanResources.Employee WHERE EmployeeID = 250 in the **AdventureWorks2012**database. When this query is executed, the query optimizer evaluates each available method for retrieving the data and selects the most efficient method. The method may be a table scan, or may be scanning one or more indexes if they exist.

When performing a table scan, the query optimizer reads all the rows in the table, and extracts the rows that meet the criteria of the query. A table scan generates many disk I/O operations and can be resource intensive. However, a table scan could be the most efficient method if, for example, the result set of the query is a high percentage of rows from the table.

When the query optimizer uses an index, it searches the index key columns, finds the storage location of the rows needed by the query and extracts the matching rows from that location. Generally, searching the index is much faster than searching the table because unlike a table, an index frequently contains very few columns per row and the rows are in sorted order.

The query optimizer typically selects the most efficient method when executing queries. However, if no indexes are available, the query optimizer must use a table scan.

**Fragmentation**

SQL Server stores data on 8KB pages. When we insert data into a table, SQL Server will allocate one page to store that data unless the data inserted is more than 8KB in which it would span multiple pages. Each page is assigned to one table. If we create 10 tables then we'll have 10 different pages.

As you insert data into a table, the data will go to the transaction log file first. The transaction log file is a sequential record meaning as you insert, update, and delete records the log will record these transactions from start to finish. The data file on the other hand is not sequential. The log file will flush the data to the data file creating pages all over the place.

## SQL Server Internal Fragmentation

SQL Server Internal Fragmentation is caused by pages that have too much free space. Let's pretend at the beginning of the day we have a table with 40 pages that are 100% full, but by the end of the day we have a table with 50 pages that are only 80% full because of various delete and insert statements throughout the day. This causes an issue because now when we need to read from this table we have to scan 50 pages instead of 40 which should may result in a decrease in performance. Let's see a quick and dirty example.

## Detecting fragmentation

I'll talk about ways to analyze fragmentation later in this tip, but for now we can right click on the index, click Properties, and Fragmentation to see fragmentation and page fullness. This is a brand new index so it's at 0% fragmentation.

Fragmentation can be easily detected by running the system function **sys.dm\_db\_index\_physical\_stats** which returns the size and the fragmentation information for the data and indexes of tables or views in SQL Server. It can be run only against a specific index in the table or view, all indexes in the specific table or view, or vs. all indexes in all databases:

USE AdventureWorks2012;

GO

-- Find the average fragmentation percentage of all indexes

-- in the HumanResources.Employee table.

SELECT a.index\_id, name, avg\_fragmentation\_in\_percent

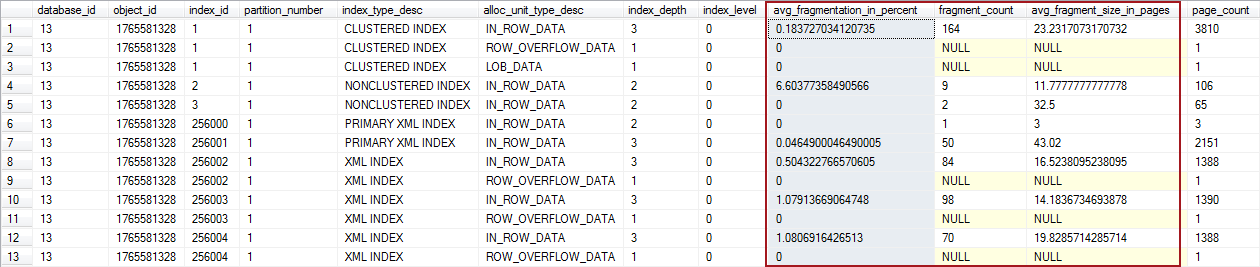
FROM sys.dm\_db\_index\_physical\_stats (DB\_ID(N'AdventureWorks2012'),

OBJECT\_ID(N'HumanResources.Employee'), NULL, NULL, NULL) AS a

JOIN sys.indexes AS b

ON a.object\_id = b.object\_id AND a.index\_id = b.index\_id;

GO



The results returned after running the procedures include following information:

* avg\_fragmentation\_in\_percent – average percent of incorrect pages in the index
* fragment\_count – number of fragments in index
* avg\_fragment\_size\_in\_pages – average number of pages in one fragment in an index

## Analyzing detection results

After the fragmentation has been detected, the next step is to determine its impact on the SQL Server and if any course of action needs to be taken.

There is no exact information on the minimal amount of fragmentation that affects the SQL Server in specific way to cause performance congestion, especially since the SQL Server environments greatly vary from one system to another.

However, there is a generally accepted solution based on the percent of fragmentation (avg\_fragmentation\_in\_percent column from the previously described sys.dm\_db\_index\_physical\_stats function)

* **Fragmentation is less than 10%** – no de-fragmentation is required. It is generally accepted that in majority of environments index fragmentation less than 10% in negligible and its performance impact on the SQL Server is minimal.
* **Fragmentation is between 10-30%** – it is suggested to perform index reorganization
* **Fragmentation is higher than 30%** – it is suggested to perform index rebuild

Here is the reasoning behind the thresholds above which will help you to determine if you should perform index rebuild or index reorganization:

Index reorganization is a process where the SQL Server goes through existing index, and cleans it up. Index rebuild is a heavy-duty process where index is deleted and then recreated from scratch with entirely new structure, free from all piled up fragments and empty-space pages.

While index reorganization is a pure cleanup operation which leaves system state as it is without locking-out affected tables and views, the rebuild process locks affected table for the whole rebuild period, which may result in long down-times that could not be acceptable in some environments.

With this in mind, it is clear that the index rebuild is a process with ‘stronger’ solution, but it comes with a price – possible long locks on affected indexed tables.

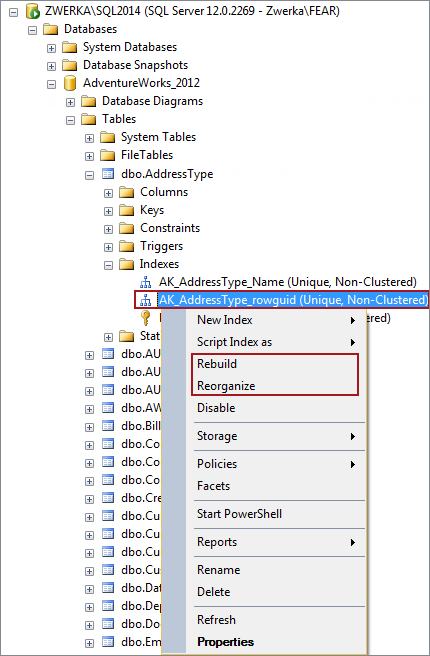
On the other side, index reorganization is a ‘lightweight’ process that will solve the fragmentation in a less effective way – since cleaned index will always be second to the new one fully made from scratch. But reorganizing index is much better from the efficiency standpoint, since it does not lock affected indexed table during the course of operation.

Servers with regular maintenance periods (e.g. regular maintenance over weekend) should almost always opt for the index rebuild, regardless of the fragmentation percent, since these environments will hardly be affected by the table lock-outs imposed by index rebuilds due to regular and long maintenance periods.

## How to reorganize and rebuild index:

**Using SQL Server Management Studio:**

1. In the Object Explorer pane navigate to and expand the SQL Server, and then the Databases node
2. Expand the specific database with fragmented index
3. Expand the Tables node, and the table with fragmented index
4. Expand the specific table
5. Expand the Indexes node
6. Right-click on the fragmented index and select **Rebuild** or **Reorganize** option in the context menu (depending on the desired action):



1. Click the **OK** button and wait for the process to complete

**Reorganize indexes in a table using Transact-SQL**

Provide appropriate database and table details and execute following code in SQL Server Management Studio to reorganize all indexes on a specific table:

USE MyDatabase;

GO

ALTER INDEX ALL ON MyTable REORGANIZE;

GO

**Rebuild indexes in a table using Transact-SQL**

Provide appropriate database and table details and execute following code in SQL Server Management Studio to rebuild all indexes on a specific table:

USE MyDatabase;

GO

ALTER INDEX ALL ON MyTable REBUILD;

GO

## SQL Server External Fragmentation

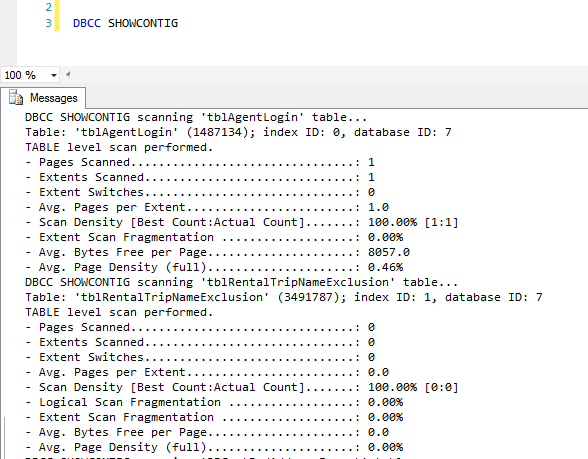
External Fragmentation is caused by pages that are out of order. Let's pretend at the beginning of the day we have a perfectly ordered table. During the day we issue hundreds of update statements possibly leaving some empty space on one page and trying to fit space into other pages. This means our storage has to jump around to obtain the data needed instead of reading in one direction.

## Analyzing SQL Server Fragmentation

So is fragmentation an issue? I believe it is. If you can store your entire database in memory or if your database is read only then I wouldn't worry about it, but most of us don't have that luxury. I've worked on thousands of servers and analyzing fragmentation levels are one of the first things I look at. In fact, just by fixing fragmentation, I've saw up to 200% improvements in query performance.

Speaking of analyzing fragmentation levels you may be wondering how we can do this. Well, there are a few ways…

**DBCC SHOWCONTIG –** this feature is old and will be removed in future versions of SQL Server, but if you're still using SQL Server 2000 or below, this will help. Instead of writing about it, I'll point you here or you can check out Chad Boyd's tip here. Both are good resources.



**sys.dm\_db\_index\_physical\_stats –** Introduced in SQL Server 2005, this dynamic management view (DMV) returns size and fragmentation information for the data and indexes of the specified table or view.

SELECT OBJECT\_NAME(ips.OBJECT\_ID)

,i.NAME

,ips.index\_id

,index\_type\_desc

,avg\_fragmentation\_in\_percent

,avg\_page\_space\_used\_in\_percent

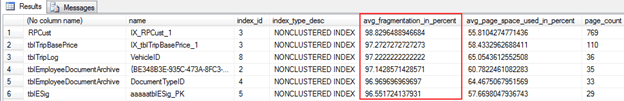
,page\_count

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'SAMPLED') ips

INNER JOIN sys.indexes i ON (ips.object\_id = i.object\_id)

AND (ips.index\_id = i.index\_id)

ORDER BY avg\_fragmentation\_in\_percent DESC



This is probably the most widely used method of analyzing fragmentation. You can see from the screenshot above that I have an index named IX\_RPCust\_1 on the RPCust table that is 98.83% fragmented. You can see more information on this DMV here from Arshad Ali.

# Pages and Extents Architecture Guide

The page is the fundamental unit of data storage in SQL Server. An extent is a collection of eight physically contiguous pages. Extents help efficiently manage pages. This guide describes the data structures that are used to manage pages and extents in all versions of SQL Server. Understanding the architecture of pages and extents is important for designing and developing databases that perform efficiently.

## Pages and Extents

The fundamental unit of data storage in SQL Server is the page. The disk space allocated to a data file (.mdf or .ndf) in a database is logically divided into pages numbered contiguously from 0 to n. Disk I/O operations are performed at the page level. That is, SQL Server reads or writes whole data pages.

Extents are a collection of eight physically contiguous pages and are used to efficiently manage the pages. All pages are stored in extents.

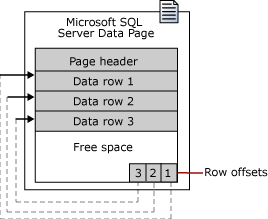
### Pages

In SQL Server, the page size is 8-KB. This means SQL Server databases have 128 pages per megabyte. Each page begins with a 96-byte header that is used to store system information about the page. This information includes the page number, page type, the amount of free space on the page, and the allocation unit ID of the object that owns the page.

The following table shows the page types used in the data files of a SQL Server database.

| **Page type** | **Contents** |
| --- | --- |
| Data | Data rows with all data, except text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data, when text in row is set to ON. |
| Index | Index entries. |
| Text/Image | Large object data types: (text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data)  Variable length columns when the data row exceeds 8 KB: (varchar, nvarchar, varbinary, and sql\_variant) |
| Global Allocation Map, Shared Global Allocation Map | Information about whether extents are allocated. |
| Page Free Space (PFS) | Information about page allocation and free space available on pages. |
| Index Allocation Map | Information about extents used by a table or index per allocation unit. |
| Bulk Changed Map | Information about extents modified by bulk operations since the last BACKUP LOG statement per allocation unit. |
| Differential Changed Map | Information about extents that have changed since the last BACKUP DATABASE statement per allocation unit. |

Data rows are put on the page serially, starting immediately after the header. A row offset table starts at the end of the page, and each row offset table contains one entry for each row on the page. Each entry records how far the first byte of the row is from the start of the page. The entries in the row offset table are in reverse sequence from the sequence of the rows on the page.



#### Large Row Support

Rows cannot span pages, however portions of the row may be moved off the row's page so that the row can actually be very large. The maximum amount of data and overhead that is contained in a single row on a page is 8,060 bytes (8-KB). However, this does not include the data stored in the Text/Image page type.

This restriction is relaxed for tables that contain varchar, nvarchar, varbinary, or sql\_variant columns. When the total row size of all fixed and variable columns in a table exceeds the 8,060-byte limitation, SQL Server dynamically moves one or more variable length columns to pages in the ROW\_OVERFLOW\_DATA allocation unit, starting with the column with the largest width.

This is done whenever an insert or update operation increases the total size of the row beyond the 8,060-byte limit. When a column is moved to a page in the ROW\_OVERFLOW\_DATA allocation unit, a 24-byte pointer on the original page in the IN\_ROW\_DATA allocation unit is maintained. If a subsequent operation reduces the row size, SQL Server dynamically moves the columns back to the original data page.

##### Row-Overflow Considerations

When you combine varchar, nvarchar, varbinary, sql\_variant, or CLR user-defined type columns that exceed 8,060 bytes per row, consider the following:

* Moving large records to another page occurs dynamically as records are lengthened based on update operations. Update operations that shorten records may cause records to be moved back to the original page in the IN\_ROW\_DATA allocation unit. Querying and performing other select operations, such as sorts or joins on large records that contain row-overflow data slows processing time, because these records are processed synchronously instead of asynchronously.  
  Therefore, when you design a table with multiple varchar, nvarchar, varbinary, sql\_variant, or CLR user-defined type columns, consider the percentage of rows that are likely to flow over and the frequency with which this overflow data is likely to be queried. If there are likely to be frequent queries on many rows of row-overflow data, consider normalizing the table so that some columns are moved to another table. This can then be queried in an asynchronous JOIN operation.
* The length of individual columns must still fall within the limit of 8,000 bytes for varchar, nvarchar, varbinary, sql\_variant, and CLR user-defined type columns. Only their combined lengths can exceed the 8,060-byte row limit of a table.
* The sum of other data type columns, including char and nchar data, must fall within the 8,060-byte row limit. Large object data is also exempt from the 8,060-byte row limit.
* The index key of a clustered index cannot contain varchar columns that have existing data in the ROW\_OVERFLOW\_DATA allocation unit. If a clustered index is created on a varchar column and the existing data is in the IN\_ROW\_DATA allocation unit, subsequent insert or update actions on the column that would push the data off-row will fail. For more information about allocation units, see Table and Index Organization.
* You can include columns that contain row-overflow data as key or nonkey columns of a nonclustered index.
* The record-size limit for tables that use sparse columns is 8,018 bytes. When the converted data plus existing record data exceeds 8,018 bytes, MSSQLSERVER ERROR 576 is returned. When columns are converted between sparse and nonsparse types, Database Engine keeps a copy of the current record data. This temporarily doubles the storage that is required for the record.
* To obtain information about tables or indexes that might contain row-overflow data, use the sys.dm\_db\_index\_physical\_stats dynamic management function.

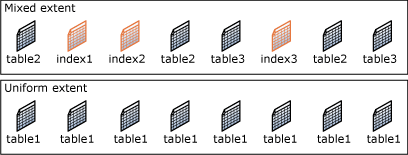
### Extents

Extents are the basic unit in which space is managed. An extent is eight physically contiguous pages, or 64 KB. This means SQL Server databases have 16 extents per megabyte.

SQL Server has two types of extents:

* **Uniform** extents are owned by a single object; all eight pages in the extent can only be used by the owning object.
* **Mixed** extents are shared by up to eight objects. Each of the eight pages in the extent can be owned by a different object.

Up to, and including, SQL Server 2014 (12.x), SQL Server does not allocate whole extents to tables with small amounts of data. A new table or index generally allocates pages from mixed extents. When the table or index grows to the point that it has eight pages, it then switches to use uniform extents for subsequent allocations. If you create an index on an existing table that has enough rows to generate eight pages in the index, all allocations to the index are in uniform extents. However, starting with SQL Server 2016 (13.x), the default for all allocations in the database is uniform extents.



## Managing Extent Allocations and Free Space

The SQL Server data structures that manage extent allocations and track free space have a relatively simple structure. This has the following benefits:

* The free space information is densely packed, so relatively few pages contain this information.  
  This increases speed by reducing the amount of disk reads that are required to retrieve allocation information. This also increases the chance that the allocation pages will remain in memory and not require more reads.
* Most of the allocation information is not chained together. This simplifies the maintenance of the allocation information.  
  Each page allocation or deallocation can be performed quickly. This decreases the contention between concurrent tasks having to allocate or deallocate pages.

### Managing Extent Allocations

SQL Server uses two types of allocation maps to record the allocation of extents:

* **Global Allocation Map (GAM)**  
  GAM pages record what extents have been allocated. Each GAM covers 64,000 extents, or almost 4 gigabytes (GB) of data. The GAM has 1-bit for each extent in the interval it covers. If the bit is 1, the extent is free; if the bit is 0, the extent is allocated.
* **Shared Global Allocation Map (SGAM)**  
  SGAM pages record which extents are currently being used as mixed extents and also have at least one unused page. Each SGAM covers 64,000 extents, or almost 4-GB of data. The SGAM has 1-bit for each extent in the interval it covers. If the bit is 1, the extent is being used as a mixed extent and has a free page. If the bit is 0, the extent is not used as a mixed extent, or it is a mixed extent and all its pages are being used.

Each extent has the following bit patterns set in the GAM and SGAM, based on its current use.

| **Current use of extent** | **GAM bit setting** | **SGAM bit setting** |
| --- | --- | --- |
| Free, not being used | 1 | 0 |
| Uniform extent, or full mixed extent | 0 | 0 |
| Mixed extent with free pages | 0 | 1 |

This causes simple extent management algorithms.

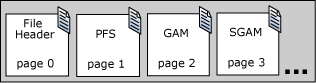
* To allocate a uniform extent, the SQL Server Database Engine searches the GAM for a 1 bit and sets it to 0.
* To find a mixed extent with free pages, the SQL Server Database Engine searches the SGAM for a 1 bit.
* To allocate a mixed extent, the SQL Server Database Engine searches the GAM for a 1 bit, sets it to 0, and then also sets the corresponding bit in the SGAM to 1.
* To deallocate an extent, the SQL Server Database Engine makes sure that the GAM bit is set to 1 and the SGAM bit is set to 0. The algorithms that are actually used internally by the SQL Server Database Engine are more sophisticated than what is described in this article, because the SQL Server Database Engine distributes data evenly in a database. However, even the real algorithms are simplified by not having to manage chains of extent allocation information.

### Tracking free space

**Page Free Space (PFS)** pages record the allocation status of each page, whether an individual page has been allocated, and the amount of free space on each page. The PFS has 1-byte for each page, recording whether the page is allocated, and if so, whether it is empty, 1 to 50 percent full, 51 to 80 percent full, 81 to 95 percent full, or 96 to 100 percent full.

After an extent has been allocated to an object, the SQL Server Database Engine uses the PFS pages to record which pages in the extent are allocated or free. This information is used when the SQL Server Database Engine has to allocate a new page. The amount of free space in a page is only maintained for heap and Text/Image pages. It is used when the SQL Server Database Engine has to find a page with free space available to hold a newly inserted row. Indexes do not require that the page free space be tracked, because the point at which to insert a new row is set by the index key values.

A PFS page is the first page after the file header page in a data file (page ID 1). This is followed by a GAM page (page ID 2), and then an SGAM page (page ID 3). There is a new PFS page approximately 8,000 pages after the first PFS page, and additional PFS pages in subsequent 8,000 page intervals. There is another GAM page 64,000 extents after the first GAM page on page 2, another SGAM page 64,000 extents after the first SGAM page on page 3, and additional GAM and SGAM pages in subsequent 64,000 extent intervals. The following illustration shows the sequence of pages used by the SQL Server Database Engine to allocate and manage extents.



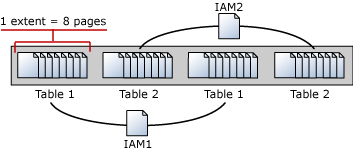
## Managing space used by objects

An **Index Allocation Map (IAM)** page maps the extents in a 4-GB part of a database file used by an allocation unit. An allocation unit is one of three types:

* IN\_ROW\_DATA  
  Holds a partition of a heap or index.
* LOB\_DATA  
  Holds large object (LOB) data types, such as xml, varbinary(max), and varchar(max).
* ROW\_OVERFLOW\_DATA  
  Holds variable length data stored in varchar, nvarchar, varbinary, or sql\_variant columns that exceed the 8,060 byte row size limit.

Each partition of a heap or index contains at least an IN\_ROW\_DATA allocation unit. It may also contain a LOB\_DATA or ROW\_OVERFLOW\_DATA allocation unit, depending on the heap or index schema.

An IAM page covers a 4-GB range in a file and is the same coverage as a GAM or SGAM page. If the allocation unit contains extents from more than one file, or more than one 4-GB range of a file, there will be multiple IAM pages linked in an IAM chain. Therefore, each allocation unit has at least one IAM page for each file on which it has extents. There may also be more than one IAM page on a file, if the range of the extents on the file allocated to the allocation unit exceeds the range that a single IAM page can record.



IAM pages are allocated as required for each allocation unit and are located randomly in the file. The system view, sys.system\_internals\_allocation\_units, points to the first IAM page for an allocation unit. All the IAM pages for that allocation unit are linked in a chain.

## Tracking Modified Extents

SQL Server uses two internal data structures to track extents modified by bulk copy operations and extents modified since the last full backup. These data structures greatly speed up differential backups. They also speed up the logging of bulk copy operations when a database is using the bulk-logged recovery model. Like the Global Allocation Map (GAM) and Shared Global Allocation Map (SGAM) pages, these structures are bitmaps in which each bit represents a single extent.

* **Differential Changed Map (DCM)**  
  This tracks the extents that have changed since the last BACKUP DATABASE statement. If the bit for an extent is 1, the extent has been modified since the last BACKUP DATABASE statement. If the bit is 0, the extent has not been modified. Differential backups read just the DCM pages to determine which extents have been modified. This greatly reduces the number of pages that a differential backup must scan. The length of time that a differential backup runs is proportional to the number of extents modified since the last BACKUP DATABASE statement and not the overall size of the database.
* **Bulk Changed Map (BCM)**  
  This tracks the extents that have been modified by bulk logged operations since the last BACKUP LOGstatement. If the bit for an extent is 1, the extent has been modified by a bulk logged operation after the last BACKUP LOG statement. If the bit is 0, the extent has not been modified by bulk logged operations. Although BCM pages appear in all databases, they are only relevant when the database is using the bulk-logged recovery model. In this recovery model, when a BACKUP LOG is performed, the backup process scans the BCMs for extents that have been modified. It then includes those extents in the log backup. This lets the bulk logged operations be recovered if the database is restored from a database backup and a sequence of transaction log backups. BCM pages are not relevant in a database that is using the simple recovery model, because no bulk logged operations are logged. They are not relevant in a database that is using the full recovery model, because that recovery model treats bulk logged operations as fully logged operations.

The interval between DCM pages and BCM pages is the same as the interval between GAM and SGAM page, 64,000 extents. The DCM and BCM pages are located behind the GAM and SGAM pages in a physical file:

