

SQL Server 索引体系结构和设计指南

2019/01/19 •   NEWBE  

本文内容

[索引设计基础知识](#)

[常规索引设计指南](#)

[元数据](#)

[聚集索引设计指南](#)

[非聚集索引设计指南](#)

[唯一索引设计指南](#)






[筛选索引设计指南](#)

[列存储索引设计指南](#)

[哈希索引设计指南](#)

[内存优化非聚集索引设计指南](#)

[其他阅读主题](#)

适用于：  SQL Server（所有支持的版本）  Azure SQL 数据库  Azure SQL 托管实例  Azure Synapse Analytics  并行数据仓库

索引设计不佳和缺少索引是提高数据库和应用程序性能的主要障碍。设计高效的索引对于获得良好的数据库和应用程序性能极为重要。在索引体系结构上的本 SQL Server 索引设计指南包含的信息和最佳做法可帮助设计满足应用程序需要的高效索引。

本指南假定读者对 SQL Server 中提供的索引类型有一般了解。有关索引类型的一般说明，请参阅 [索引类型](#)。

本指南涉及以下类型的索引：

- 聚集
- 非聚集
- 唯一
- Filtered
- 列存储
- 哈希
- 内存优化非聚集索引

有关 XML 索引的信息，请参阅 [XML 索引概述](#)和[选择性 XML 索引 \(SXI\)](#)。

有关空间索引的信息，请参阅[空间索引概述](#)。

有关全文索引的信息，请参阅[填充全文索引](#)。

索引设计基础知识

请考虑普通书籍：本书末尾有一个索引，可帮助快速查找书籍内的信息。索引是按顺序排列的关键字列表，每个关键字旁边是一组页码，这些页码指向可在其中找到每个关键字的页面。SQL Server 索引也一样：它是按顺序排列的值列表，每个值都有指向这些值所在的数据**页面**的指针。索引本身存储在页面上，构成 SQL Server 中的索引页面。在普通书籍中，如果索引跨越多个页面并且必须找到指向包含单词“SQL”的所有页面的指针，则必须翻阅，直到找到包含关键字“SQL”的索引页面。在索引页面，你可以找到指向所有书籍页面的指针。如果在索引的开头创建了一个页面，其中包含可以找到每个字母的字母顺序列表，则可以进一步优化此页面。例如：“A 到 D - 第 121 页”，“E 到 G - 第 122 页”等等。这个额外的页面将使你不必执行翻阅索引才能找到起始位置的步骤。此类页面在常规书籍中不存在，但在 SQL Server 索引中确实存在。此单个页面称为索引的根页。根页是 SQL Server 索引使用的树结构的起始页面。按照树的类比，包含指向实际数据的指针的结束页面被称为树的“叶页”。

SQL Server 索引是与表或视图关联的磁盘上或内存中结构，可以加快从表或视图中检索行的速度。索引包含由表或视图中的一列或多列生成的键。对于磁盘上的索引，这些键以树结构（B 树）存储，使 SQL Server 可以快速高效地找到与键值关联的一行或多行。

索引在逻辑上以组织为包含行和列的表存储数据；在物理上以按行数据格式（称为行存储¹），或以按列数据格式（称为**列存储**）存储数据。

为数据库及其工作负荷选择正确的索引是一项需要在查询速度与更新所需开销之间取得平衡的复杂任务。如果索引较窄，或者说索引关键字中只有很少的几列，则需要的磁盘空间和维护开销都较少。而另一方面，宽索引可覆盖更多的查询。您可能需要试验若干不同的设计，才能找到最有效的索引。可以添加、修改和删除索引而不影响数据库架构或应用程序设计。因此，应试验多个不同的索引而无需犹豫。

SQL Server 中的查询优化器可在大多数情况下可靠地选择最高效的索引。总体索引设计策略应为查询优化器提供可供选择的多个索引，并依赖查询优化器做出正确的决定。这在多种情况下可减少分析时间并获得良好的性能。若要查看查询优化器对特定查询使用的索引，请在 SQL Server Management Studio 中的“查询”菜单上选择“包括实际的执行计划”。

不要总是将索引的使用等同于良好的性能，或者将良好的性能等同于索引的高效使用。如果只要使用索引就能获得最佳性能，那查询优化器的工作就简单了。但事实上，不正确的索引选择并不能获得最佳性能。因此，查询优化器的任务是只在索引或索引组合能提高性能时才选择它，而在索引检索有碍性能时则避免使用它。

¹ 行存储是存储关系表数据的传统方法。在 SQL Server 中，行存储是指基础数据存储格式为堆、B 树（**聚集索引**）或内存优化表的表。

索引设计任务

建议的索引设计策略包括以下任务：

- 例如，数据库是否是频繁修改数据的联机事务处理 (OLTP) 数据库，必须可承受高吞吐量。从 SQL Server 2014 (12.x) 开始，内存优化表和索引提供无锁设计，尤其适用于此应用场景。有关详细信息，请参阅本指南中的[内存优化表的索引](#)或[内存优化表的非聚集索引设计指南](#)和[内存优化表的哈希索引设计指南](#)。
 - 或者，数据库是否是一种决策支持系统 (DSS) 或数据仓库 (OLAP) 数据库，必须快速处理超大型数据集。从 SQL Server 2012 (11.x) 开始，列存储索引尤其适用于典型的数据仓库数据集。列存储索引可以通过为常见数据仓库查询（如筛选、聚合、分组和星型联接查询）提供更快的性能，以转变用户的数据仓库体验。有关详细信息，请参阅本指南中的[列存储索引概述](#)或[列存储索引设计指南](#)。
2. 了解最常用的查询的特征。例如，了解到最常用的查询联接两个或多个表将有助于决定要使用的最佳索引类型。
 3. 了解查询中使用的列的特征。例如，某个索引对于含有整数数据类型同时还是唯一的或非空的列是理想索引。对于具有定义完善的数据子集的列，您可以在 SQL Server 2008 和更高版本中使用筛选索引。有关详细信息，请参阅本指南中的[筛选索引设计指南](#)。
 4. 确定哪些索引选项可在创建或维护索引时提高性能。例如，对某个现有大型表创建聚集索引将会受益于 `ONLINE` 索引选项。ONLINE 选项允许在创建索引或重新生成索引时继续对基础数据执行并发活动。有关详细信息，请参阅[设置索引选项](#)。
 5. 确定索引的最佳存储位置。非聚集索引可以与基础表存储在同一个文件组中，也可以存储在不同的文件组中。索引的存储位置可通过提高磁盘 I/O 性能来提高查询性能。例如，将非聚集索引存储在表文件组所在磁盘以外的某个磁盘上的一个文件组中可以提高性能，因为可以同时读取多个磁盘。
或者，聚集索引和非聚集索引也可以使用跨越多个文件组的分区方案。在维护整个集合的完整性时，使用分区可以快速而有效地访问或管理数据子集，从而使大型表或索引更易于管理。有关详细信息，请参阅[Partitioned Tables and Indexes](#)。在考虑分区时，应确定是否应对齐索引，即，是按实质上与表相同的方式进行分区，还是单独分区。

常规索引设计指南

经验丰富的数据库管理员能够设计出好的索引集，但是，即使对于不特别复杂的数据库和工作负荷来说，这项任务也十分复杂、耗时和易于出错。了解数据库、查询和数据列的特征可以帮助您设计出最佳索引。

数据库注意事项

设计索引时，应考虑以下数据库准则：

- 对表编制大量索引会影响 `INSERT`、`UPDATE`、`DELETE` 和 `MERGE` 语句的性能，因为当表中的数据更改时，所有索引都须适当调整。例如，如果在多个索引中使用了某个列，并且执行了修改该列数据的 `UPDATE` 语句，则必须更新包含该列的每个索引以及基础的基表（堆或聚集索引）中的该列。
 - 避免对经常更新的表进行过多的索引，并且索引应保持较窄，就是说，列要尽可能少。
 - 使用多个索引可以提高更新少而数据量大的查询的性能。大量索引可以提高不修改数据的查询（例如 `SELECT` 语句）的性能，因为查询优化器有更多的索引可供选择，从而可以确定最快的访问方法。
- 对小表进行索引可能不会产生优化效果，因为查询优化器在遍历用于搜索数据的索引时，花费的时间可能比执行简单的表扫描还长。因此，小表的索引可能从来不用，但仍必须在表中的数据更改时进行维护。
- 视图包含聚合、表联接或聚合和联接的组合时，视图的索引可以显著地提升性能。若要使查询优化器使用视图，并不一定非要在查询中显式引用该视图。
- 使用数据库引擎优化顾问来分析数据库并生成索引建议。有关详细信息，请参阅 [Database Engine Tuning Advisor](#)。

查询注意事项

设计索引时，应考虑以下查询准则：

- 为经常用于查询中的谓词和联接条件的列创建非聚集索引。这些是你的 `SARGable`¹ 列。但是，应避免添加不必要的列。添加太多索引列可能对磁盘空间和索引维护性能产生负面影响。
- 涵盖索引可以提高查询性能，因为符合查询要求的全部数据都存在于索引本身中。也就是说，只需要索引页，而不需要表的数据页或聚集索引来检索所需数据，因此，减少了总体磁盘 I/O。例如，对某一表（其中对列 `a`、`b` 和 `c` 创建了组合索引）的列 `a` 和 `b` 的查询，仅仅从该索引本身就可以检索指定数据。

❶ 重要

覆盖索引是针对**非聚集索引**的指定，它直接解析一个或几个类似的查询结果，而不访问其基表，并且不会引发查找。此类索引在叶级别上具有所有必要的非 `SARGable` 列。这意味着，由 `SELECT` 子句以及所有 `WHERE` 和 `JOIN` 参数返回的列都被索引所覆盖。当与表本身的行和列相比，如果索引足够窄，那么执行查询的 I/O 可能会少得多，这意味着它是总列的一个真正子集。如果选择大型

表的一小部分，请考虑覆盖索引，其中的小部分是由一个固定谓词定义，比如一个**稀疏列**，例如它只包含几个非 NULL 值。

- 将插入或修改尽可能多的行的查询写入单个语句内，而不要使用多个查询更新相同的行。仅使用一个语句，就可以利用优化的索引维护。
- 评估查询类型以及如何在查询中使用列。例如，在完全匹配查询类型中使用的列就适合用于非聚集索引或聚集索引。

¹ “SARGable”一词在关系数据库中指的是一个搜索可论证的谓词，它可以利用一个索引来加快查询的执行过程。

列注意事项

设计索引时，应考虑以下列准则：

- 对于聚集索引，请保持较短的索引键长度。另外，对唯一列或非空列创建聚集索引可以使聚集索引获益。
- 无法指定 `ntext`、`text`、`image`、`varchar(max)`、`nvarchar(max)` 和 `varbinary(max)` 数据类型的列为索引键列。不过，`varchar(max)`、`nvarchar(max)`、`varbinary(max)` 和 `xml` 数据类型的列可以作为非键索引列参与非聚集索引。有关详细信息，请参阅本指南中的 [具有包含列的索引](#)。
- `xml` 数据类型的列只能在 XML 索引中用作键列。有关详细信息，请参阅 [XML 索引 \(SQL Server\)](#)。SQL Server 2012 SP1 引入了称作选择性 XML 索引的一种新的 XML 索引。这个新的索引可提高 SQL Server 中针对作为 XML 存储的数据的查询性能，从而通过降低索引本身的存储成本来加快大型 XML 数据工作负荷的索引编制和改进可伸缩性。有关详细信息，请参阅[选择性 XML 索引 \(SXI\)](#)。
- 检查列的唯一性。在同一个列组合的唯一索引而不是非唯一索引提供了有关使索引更有用的查询优化器的附加信息。有关详细信息，请参阅本指南中的 [唯一索引设计指南](#)。
- 在列中检查数据分布。通常情况下，为包含很少唯一值的列创建索引或在这样的列上执行联接将导致长时间运行的查询。这是数据和查询的基本问题，通常不识别这种情况就无法解决这类问题。例如，如果物理电话簿按姓的字母顺序排序，而城市里所有人的姓都是 Smith 或 Jones，则无法快速找到某个人。有关数据分布的详细信息，请参阅 [统计信息](#)。
- 考虑对具有定义完善的子集的列（例如，稀疏列、大部分值为 NULL 的列、含各类值的列以及含不同范围的值的列）使用筛选索引。设计良好的筛选索引可以提高查询性能，降低索引维护成本和存储成本。

- 如果索引包含多列，则应牢记列的顺序。用于等于(=)、大于(>)、小于(<)或 BETWEEN 搜索条件的 WHERE 子句或者参与联接的列应该放在最前面。其他列应该基于其非重复级别进行排序，就是说，从最不重复的列到最重复的列。

例如，如果将索引定义为 LastName、FirstName，则该索引在搜索条件为 WHERE LastName = 'Smith' 或 WHERE LastName = Smith AND FirstName LIKE 'J%' 时将很有用。不过，查询优化器不会将此索引用于基于 FirstName (WHERE FirstName = 'Jane') 而搜索的查询。

- 考虑对计算列进行索引。有关详细信息，请参阅 [计算列上的索引](#)。

索引的特征

在确定某一索引适合某一查询之后，可以选择最适合具体情况的索引类型。索引包含以下特性：

- 聚集还是非聚集
- 唯一还是非唯一
- 单列还是多列
- 索引中的列是升序排序还是降序排序
- 非聚集索引是全表还是经过筛选
- 列存储与行存储
- 内存优化表的哈希索引与非聚集索引

您也可以通过设置选项（例如 FILLFACTOR）自定义索引的初始存储特征以优化其性能或维护。而且，通过使用文件组或分区方案可以确定索引存储位置来优化性能。

文件组或分区方案的索引设置

开发索引设计策略时，应该考虑在与数据库相关联的文件组上放置索引。仔细选择文件组或分区方案可以改进查询性能。

默认情况下，索引存储在基表所在的文件组上，该索引即在该基表上创建。非分区聚集索引和基表始终在同一个文件组中。但是，您可以执行以下操作：

- 为除基表或聚集索引的文件组之外的文件组创建非聚集索引。
- 对要涵盖多个文件组的聚集和非聚集索引进行分区。
- 通过删除聚集索引并在 DROP INDEX 语句的 MOVE TO 子句中指定新的文件组或分区方案，或者在 CREATE INDEX 语句中使用 DROP_EXISTING 子句，将表从一个文件组移至另一个文件组。

通过对其他文件组创建非聚集索引，可以在文件组通过自带的控制器使用不同的物理驱动器时实现性能提升。这样一来，数据和索引信息即可由多个磁头并行读取。例如，如

果文件组 `Table_A` 的 `f1` 和文件组 `Index_A` 的 `f2` 都由同一个查询使用，就可无争夺地充分使用这两个文件组，因此可以实现性能提升。但是，如果 `Table_A` 由查询扫描而没有引用 `Index_A`，则仅使用文件组 `f1`。这不会引起性能提升。

由于无法预测将要发生的访问类型以及访问时间，因此更好的办法可能是展开所有文件组中的表和索引。这将保证能够访问所有磁盘，因为所有数据和索引在所有磁盘上均匀展开，不受访问数据的方式的限制。这对系统管理员来说也是更简单的方法。

在多个文件组中分区

还可以考虑在多个文件组中对聚集和非聚集索引分区。根据分区函数，对已分区的索引进行水平分区或按行分区。分区函数定义如何根据某些列（称为分区依据列）的值将每一行映射到一组分区。分区方案将分区映射指定给一组文件组。

对索引进行分区有以下优点：

- 提供使大型索引更易管理的可伸缩系统。例如，OLTP 系统可以实现处理大型索引的可识别分区的应用程序。
- 使查询运行得更快、更有效。当查询访问索引的几个分区时，查询优化器同时可以处理各个分区，但不包括不受该查询影响的分区。


有关详细信息，请参阅 [Partitioned Tables and Indexes](#)。

索引排序顺序设计指南

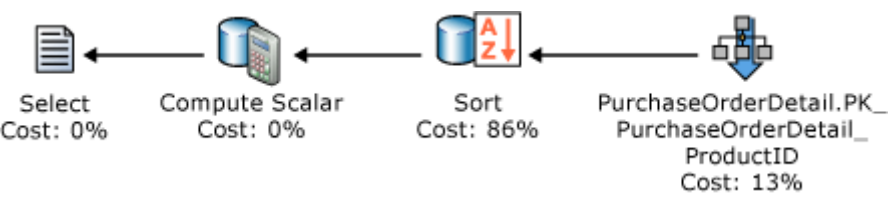
定义索引时，应该考虑索引键列的数据是按升序还是按降序存储。升序是默认设置，保持与 SQL Server 早期版本的兼容性。CREATE INDEX、CREATE TABLE 和 ALTER TABLE 语句的语法在索引和约束中的各列上支持关键字 ASC（升序）和 DESC（降序）：

当引用表的查询包含用以指定索引中键列的不同方向的 ORDER BY 子句时，指定键值存储在该索引中的顺序很有用。在这些情况下，索引就无需在查询计划中使用 SORT 运算符。因此，使得查询更有效。例如，Adventure Works Cycles 采购部门的买方不得不评估他们从供应商处购买的产品的质量。买方倾向于查验那些由具有高拒绝率的供应商发送的产品。检索数据以满足此条件需要将 `RejectedQty` 表中的

`Purchasing.PurchaseOrderDetail` 列按降序（由大到小）排序，并且将 `ProductID` 列按升序（由小到大）排序，如下列查询所示。

SQL	 复制
<pre>SELECT RejectedQty, ((RejectedQty/OrderQty)*100) AS RejectionRate, ProductID, DueDate FROM Purchasing.PurchaseOrderDetail ORDER BY RejectedQty DESC, ProductID ASC;</pre>	

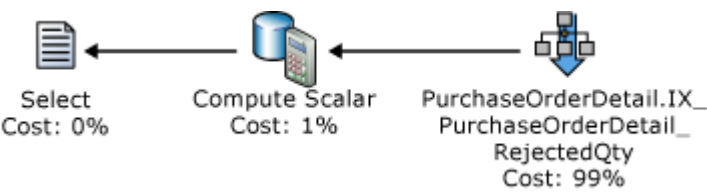
此查询的下列执行计划显示了查询优化器使用 SORT 运算符按 ORDER BY 子句指定的顺序返回结果集。



如果使用与查询的 ORDER BY 子句中的键列匹配的键列创建索引，则无需在查询计划中使用 SORT 运算符，从而使查询计划更有效。

SQL	复制
<pre>CREATE NONCLUSTERED INDEX IX_PurchaseOrderDetail_RejectedQty ON Purchasing.PurchaseOrderDetail (RejectedQty DESC, ProductID ASC, DueDate, OrderQty);</pre>	

再次执行查询后，下列执行计划显示未使用 SORT 运算符，而使用了新创建的非聚集索引。



数据库引擎 可以在两个方向上同样有效地移动。 对于一个在 ORDER BY 子句中列的排序方向倒排的查询， 仍然可以使用定义为 (RejectedQty DESC, ProductID ASC) 的索引。 例如， 包含 ORDER BY 子句 ORDER BY RejectedQty ASC, ProductID DESC 的查询可以使用该索引。

只能为索引中的键列指定排序顺序。 [sys.index_columns](#) 目录视图和 INDEXKEY_PROPERTY 函数报告索引列是按升序还是降序存储。

元数据

使用这些元数据视图可以查看索引的属性。 其他体系结构信息嵌入在其中的某些视图中。

① 备注

对于列存储索引，所有列在元数据中作为包含性列存储。 列存储索引中没有任何键列。



[sys.indexes \(Transact-SQL\)](#)

[sys.index_columns \(Transact-SQL\)](#)

[sys.partitions \(Transact-SQL\)](#)

[sys.internal_partitions \(Transact-SQL\)](#)

[sys.dm_db_index_operational_stats \(Transact-SQL\)](#)

[sys.dm_db_index_physical_stats \(Transact-SQL\)](#)

[sys.column_store_segments \(Transact-SQL\)](#)

[sys.column_store_dictionaries \(Transact-SQL\)](#)

[sys.column_store_row_groups \(Transact-SQL\)](#)

[sys.dm_db_column_store_row_group_operational_stats \(Transact-SQL\)](#)

[sys.dm_db_column_store_row_group_physical_stats \(Transact-SQL\)](#)

[sys.dm_column_store_object_pool \(Transact-SQL\)](#)

[sys.dm_db_column_store_row_group_operational_stats \(Transact-SQL\)](#)

[sys.dm_db_xtp_hash_index_stats \(Transact-SQL\)](#)

[sys.dm_db_xtp_index_stats \(Transact-SQL\)](#)

[sys.dm_db_xtp_object_stats \(Transact-SQL\)](#)

[sys.dm_db_xtp_nonclustered_index_stats \(Transact-SQL\)](#)

[sys.dm_db_xtp_table_memory_stats \(Transact-SQL\)](#)

[sys.hash_indexes \(Transact-SQL\)](#)

[sys.memory_optimized_tables_internal_attributes \(Transact-SQL\)](#)

聚集索引设计指南

聚集索引基于数据行的键值在表内排序和存储这些数据行。每个表只能有一个聚集索引，因为数据行本身只能按一个顺序存储。每个表几乎都对列定义聚集索引来实现下列功能：

- 可用于经常使用的查询。
- 提供高度唯一性。

ⓘ 备注

创建 PRIMARY KEY 约束时，将在列上自动创建唯一索引。默认情况下，此索引是聚集索引，但是在创建约束时，可以指定创建非聚集索引。

- 可用于范围查询。

如果未使用 `UNIQUE` 属性创建聚集索引，数据库引擎会自动向表添加一个 4 字节的唯一标识符列。必要时，数据库引擎 将向行自动添加一个唯一标识符值以使每个键唯一。此列和列值供内部使用，用户不能查看或访问。

聚集索引体系结构

在 SQL Server 中，索引是按 B 树结构组织的。索引 B 树中的每一页称为一个索引节点。B 树的顶端节点称为根节点。索引中的底层节点称为叶节点。根节点与叶节点之间的任何索引级别统称为中间级。在聚集索引中，叶节点包含基础表的数据页。根节点和中间级节点包含存有索引行的索引页。每个索引行包含一个键值和一个指针，该指针指向 B 树上的某一中间级页或叶级索引中的某个数据行。每级索引中的页均被链接在双向链接列表中。

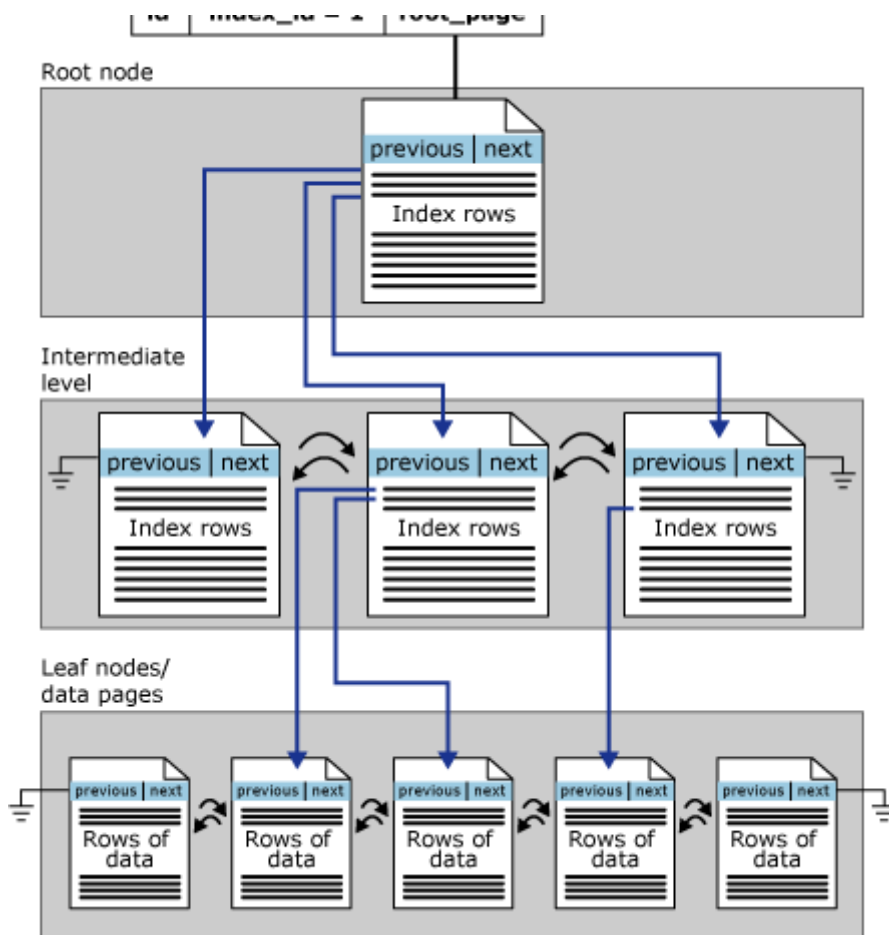
聚集索引在 `sys.partitions` 中有一行，其中，索引使用的每个分区的 `index_id = 1`。默认情况下，聚集索引有单个分区。当聚集索引有多个分区时，每个分区都有一个包含该特定分区相关数据的 B 树结构。例如，如果聚集索引有四个分区，就有四个 B 树结构，每个分区中有一个 B 树结构。

根据聚集索引中的数据类型，每个聚集索引结构将有一个或多个分配单元，将在这些单元中存储和管理特定分区的相关数据。每个聚集索引的每个分区中至少有一个 `IN_ROW_DATA` 分配单元。如果聚集索引包含大型对象 (LOB) 列，则它的每个分区中还会有一个 `LOB_DATA` 分配单元。如果聚集索引包含的变量长度列超过 8,060 字节的行大小限制，则它的每个分区中还会有一个 `ROW_OVERFLOW_DATA` 分配单元。

数据链内的页和行将按聚集索引键值进行排序。所有插入操作都在所插入行中的键值与现有行中的排序顺序相匹配时执行。

下图显式了聚集索引单个分区中的结构。

id	index_id = 1	root page
----	--------------	-----------



查询注意事项

在创建聚集索引之前，应先了解数据是如何被访问的。考虑对具有以下特点的查询使用聚集索引：

- 使用运算符（如 `BETWEEN`、`>`、`>=`、`<` 和 `<=`）返回一系列值。

使用聚集索引找到包含第一个值的行后，便可以确保包含后续索引值的行物理相邻。例如，如果某个查询在一系列销售订单号间检索记录，`SalesOrderNumber` 列的聚集索引可快速定位包含起始销售订单号的行，然后检索表中所有连续的行，直到检索到最后的销售订单号。

- 返回大型结果集。
- 使用 `JOIN` 子句；一般情况下，使用该子句的是外键列。
- 使用 `ORDER BY` 或 `GROUP BY` 子句。

在 `ORDER BY` 或 `GROUP BY` 子句中指定的列的索引，可以使数据库引擎不必对数据进行排序，因为这些行已经排序。这会有助于提升查询性能。

列注意事项

一般情况下，定义聚集索引键时使用的列越少越好。考虑具有下列一个或多个属性的列：

- 唯一或包含许多不重复的值

例如，雇员 ID 唯一地标识雇员。`EmployeeID` 列的聚集索引或主键约束可提高基于雇员 ID 号搜索雇员信息的查询的性能。另外，可对 `LastName`、`FirstName`、`MiddleName` 列创建聚集索引，因为经常以这种方式分组和查询雇员记录，而且这些列的组合还可提供高区分度。

💡 提示

如果没有另行指定，在创建主键约束时，SQL Server 会创建一个聚集索引来支持该约束。虽然可使用 `uniqueidentifier` 来强制实施作为主键的唯一性，但它不是有效的聚集键。如果使用 `uniqueidentifier` 作为主键，建议将其创建为非聚集索引，然后使用另一列（如 `IDENTITY`）创建聚集索引。

- 按顺序被访问

例如，产品 ID 唯一地标识 `Production.Product` 数据库的 `AdventureWorks2012` 表中的产品。在其中指定顺序搜索的查询（如 `WHERE ProductID BETWEEN 980 and 999`）将从 `ProductID` 的聚集索引受益。这是因为行将按该键列的排序顺序存储。

- 定义为 `IDENTITY`。
- 经常用于对表中检索到的数据进行排序。

按该列对表进行聚集（即物理排序）是一个好方法，它可以在每次查询该列时节省排序操作的成本。

聚集索引不适用于具有下列属性的列：

- 频繁更改的列

这将导致整行移动，因为数据库引擎必须按物理顺序保留行中的数据值。这一点要特别注意，因为在大容量事务处理系统中数据通常是可变的。

- 宽键

宽键是若干列或若干大型列的组合。所有非聚集索引将聚集索引中的键值用作查找键。为同一表定义的任何非聚集索引都将增大许多，这是因为非聚集索引项包含聚集键，同时也包含为此非聚集索引定义的键列。

非聚集索引设计指南

非聚集索引包含索引键值和指向表数据存储位置的行定位器。可以对表或索引视图创建多个非聚集索引。通常，设计非聚集索引是为改善经常使用的、没有建立聚集索引的查询的性能。

与使用书中索引的方式相似，查询优化器在搜索数据值时，先搜索非聚集索引以找到数据值在表中的位置，然后直接从该位置检索数据。这使非聚集索引成为完全匹配查询的最佳选择，因为索引包含说明查询所搜索的数据值在表中的精确位置的项。例如，为了从 `HumanResources.Employee` 表中查询向特定经理负责的所有雇员，查询优化器可能使用非聚集索引 `IX_Employee_ManagerID`；它以 `ManagerID` 作为其键列。查询优化器能快速找出索引中与指定 `ManagerID` 匹配的所有项。每个索引项都指向表或聚集索引中准确的页和行，其中可以找到相应的数据。在查询优化器在索引中找到所有项之后，它可以直接转到准确的页和行进行数据检索。

非聚集索引体系结构

非聚集索引与聚集索引具有相同的 B 树结构，它们之间的显著差别在于以下两点：

- 基础表的数据行不按非聚集键的顺序排序和存储。
- 非聚集索引的叶级别是由索引页而不是由数据页组成。

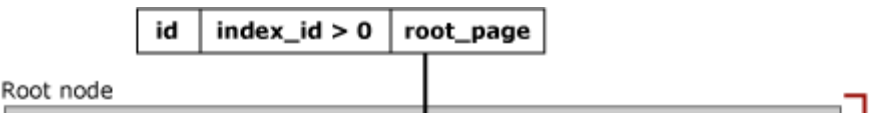
非聚集索引行中的行定位器或是指向行的指针，或是行的聚集索引键，如下所述：

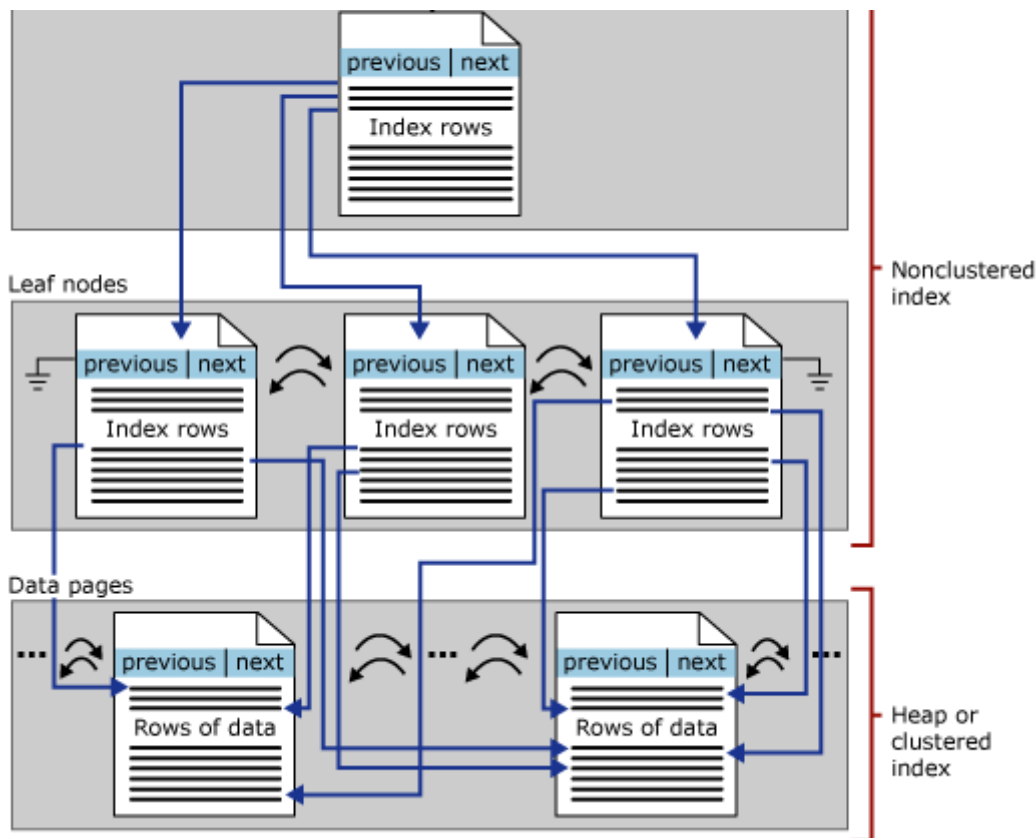
- 如果表是堆（意味着该表没有聚集索引），则行定位器是指向行的指针。该指针由文件标识符 (ID)、页码和页上的行数生成。整个指针称为行 ID (RID)。
- 如果表有聚集索引或索引视图上有聚集索引，则行定位器是行的聚集索引键。

对于索引使用的每个分区，非聚集索引在 `index_id > 1` 的 `sys.partitions` 中都有对应的一行。默认情况下，一个非聚集索引有单个分区。如果一个非聚集索引有多个分区，则每个分区都有一个包含该特定分区的索引行的 B 树结构。例如，如果一个非聚集索引有四个分区，那么就有四个 B 树结构，每个分区中一个。

根据非聚集索引中数据类型的不同，每个非聚集索引结构会有一个或多个分配单元，在其中存储和管理特定分区的数据。每个非聚集索引至少有一个针对每个分区的 `IN_ROW_DATA` 分配单元（存储索引 B 树页）。如果非聚集索引包含大型对象 (LOB) 列，则还有一个针对每个分区的 `LOB_DATA` 分配单元。此外，如果非聚集索引包含的可变长度列超过 8,060 字节的行大小限制，则还有一个针对每个分区的 `ROW_OVERFLOW_DATA` 分配单元。

下图说明了单个分区中的非聚集索引结构。





数据库注意事项

设计非聚集索引时需要注意数据库的特征。

- 更新要求较低但包含大量数据的数据库或表可以从许多非聚集索引中获益从而改善查询性能。与全表非聚集索引相比，考虑为定义完善的数据子集创建筛选索引可以提高查询性能、降低索引存储开销并减少索引维护开销。

决策支持系统应用程序和主要包含只读数据的数据库可以从许多非聚集索引中获益。查询优化器具有更多可供选择的索引用来确定最快的访问方法，并且数据库的低更新特征意味着索引维护不会降低性能。

- 联机事务处理 (OLTP) 应用程序和包含经常更新的表的数据库应避免过多索引。此外，索引应该是窄的，即列越少越好。

对表编制大量索引会影响 INSERT、UPDATE、DELETE 和 MERGE 语句的性能，因为当表中的数据更改时，所有索引都须进行适当的调整。

查询注意事项

在创建非聚集索引之前，应先了解访问数据的方式。考虑对具有以下属性的查询使用非聚集索引：

- 使用 `JOIN` 或 `GROUP BY` 子句。

应力联按和分组操作中所涉及的列创建多个非聚集索引，为任何外键列创建一个聚集索引。

- 不返回大型结果集的查询。

创建筛选索引以覆盖从大型表中返回定义完善的行子集的查询。

💡 提示

通常，CREATE INDEX 语句的 WHERE 子句匹配所覆盖的查询的 WHERE 子句。

- 包含经常包含在查询的搜索条件（例如返回完全匹配的 WHERE 子句）中的列。

💡 提示

添加新索引时，请考虑成本和权益。将其他查询需求合并到现有索引中可能更可取。例如，如果允许覆盖多个关键查询，而不是每个关键查询都有一个覆盖索引，则考虑添加一个或两个额外的叶级别列到现有索引。

列注意事项

考虑具有以下一个或多个属性的列：

- 覆盖查询。

当索引包含查询中的所有列时，性能可以提升。查询优化器可以找到索引内的所有列值；不会访问表或聚集索引数据，这样就减少了磁盘 I/O 操作。使用具有**包含列**的索引来添加覆盖列，而不是创建宽索引键。

如果表有聚集索引，则该聚集索引中定义的列将自动追加到表上每个非聚集索引的末端。这可以生成覆盖查询，而不用在非聚集索引定义中指定聚集索引列。例如，如果一个表在 **C** 列上有聚集索引，则 **B** 和 **A** 列的非聚集索引将具有其自己的键值列 **B**、**A** 和 **C**。

- 大量非重复值，如姓氏和名字的组合（前提是聚集索引被用于其他列）。

如果只有很少的非重复值，例如仅有 1 和 0，则大多数查询将不使用索引，因为此时表扫描通常更有效。对于这种类型的数据，应考虑对仅出现在少数行中的非重复值创建筛选索引。例如，如果大部分值都是 0，则查询优化器可以对包含 1 的数据行使用筛选查询。

使用包含列扩展非聚集索引

您可以使用包含列扩展非聚集索引的性能。扩展非聚集索引的性能。通过包含非键

您可以通过将非键列添加到非聚集索引的叶级，扩展非聚集索引的功能。通过包含非键列，可以创建覆盖更多查询的非聚集索引。这是因为非键列具有下列优点：

- 它们可以是不允许作为索引键列的数据类型。
- 在计算索引键列数或索引键大小时，数据库引擎 不考虑它们。

当查询中的所有列都作为键列或非键列包含在索引中时，带有包含性非键列的索引可以显著提高查询性能。这样可以实现性能提升，因为查询优化器可以在索引中找到所有列值；不访问表或聚集索引数据，从而减少磁盘 I/O 操作。

❗ 备注

当索引包含查询引用的所有列时，它通常称为“覆盖查询”。

键列存储在索引的所有级别中，而非键列仅存储在叶级别中。


使用包含列以避免大小限制

可以将非键列包含在非聚集索引中，以避免超过当前索引大小的限制（最大键列数为 16，最大索引键大小为 900 字节）。数据库引擎 计算索引键列数或索引键大小时，不考虑非键列。

例如，假设要为 `Document` 表中的以下列建立索引：

- `Title nvarchar(50)`
- `Revision nchar(5)`
- `FileName nvarchar(400)`

因为 `nchar` 和 `nvarchar` 数据类型的每个字符需要 2 个字节，所以包含这三列的索引将超出 900 字节的大小限制 10 个字节 ($455 * 2$)。使用 `INCLUDE` 语句的 `CREATE INDEX` 子句，可以将索引键定义为 (`Title`, `Revision`)，将 `FileName` 定义为非键列。这样，索引键大小将为 110 个字节 ($55 * 2$)，并且索引仍将包含所需的所有列。下面的语句就创建了这样的索引。

SQL	 复制
<pre>CREATE INDEX IX_Document_Title ON Production.Document (Title, Revision) INCLUDE (FileName);</pre>	

带有包含列的索引准则

对于包含列的非聚集索引，请考虑下列准则。

设计带有包含列的非聚集索引时，请考虑下列准则。

- 在 CREATE INDEX 语句的 INCLUDE 子句中定义非键列。
- 只能对表或索引视图的非聚集索引定义非键列。
- 允许除 text、ntext 和 image 之外的所有数据类型。
- 精确或不精确的确定性计算列都可以是包含列。有关详细信息，请参阅 [计算列上的索引](#)。
- 与键列一样，只要允许将计算列数据类型作为非键索引列，从 image、ntext 和 text 数据类型派生的计算列就可以作为非键（包含性）列。
- 不能同时在 INCLUDE 列表和键列列表中指定列名。
- INCLUDE 列表中的列名不能重复。

列大小准则

- 必须至少定义一个键列。最大非键列数为 1023 列。也就是最大的表列数减 1。
- 索引键列（不包括非键）必须遵守现有索引大小的限制（最大键列数为 16，总索引键大小为 900 字节）。
- 所有非键列的总大小只受 INCLUDE 子句中所指定列的大小限制；例如，varchar(max) 列限制为 2 GB。

列修改准则

修改已定义为包含列的表列时，要受下列限制：

- 除非先删除索引，否则无法从表中删除非键列。
- 除进行下列更改外，不能对非键列进行其他更改：
 - 将列的为空性从 NOT NULL 改为 NULL。
 - 增加 varchar、nvarchar 或 varbinary 列的长度。

❗ 备注


这些列修改限制也适用于索引键列。

设计建议

避免在大型索引键上添加小的非聚集索引。以便只有用于搜索和查找的列是键列。收集并


重新设计索引键入小较入的非聚集索引，以便它有利于搜索和查找的列为键列。 将覆盖查询的所有其他列设置为包含性非键列。 这样，将具有覆盖查询所需的所有列，但索引键本身较小，而且效率高。

例如，假设要设计覆盖下列查询的索引。

SQL	 复制
<pre>SELECT AddressLine1, AddressLine2, City, StateProvinceID, PostalCode FROM Person.Address WHERE PostalCode BETWEEN N'98000' and N'99999';</pre>	

若要覆盖查询，必须在索引中定义每列。 尽管可以将所有列定义为键列，但键大小为 334 字节。 因为实际上用作搜索条件的唯一列是 `PostalCode` 列（长度为 30 字节），所以更好的索引设计应该将 `PostalCode` 定义为键列并包含作为非键列的所有其他列。

下面的语句创建了一个覆盖查询的带有包含列的索引。

SQL	 复制
<pre>CREATE INDEX IX_Address_PostalCode ON Person.Address (PostalCode) INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);</pre>	

性能注意事项

避免添加不必要的列。 添加过多的索引列（键列或非键列）会对性能产生下列影响：

- 一页上能容纳的索引行将更少。 这样会使 I/O 增加并降低缓存效率。
- 需要更多的磁盘空间来存储索引。 特别是，将 `varchar(max)`、`nvarchar(max)`、`varbinary(max)` 或 `xml` 数据类型添加为非键索引列会显著增加磁盘空间要求。 这是因为列值被复制到了索引叶级别。 因此，它们既驻留在索引中，也驻留在基表中。
- 索引维护可能会增加对基础表或索引视图执行修改、插入、更新或删除操作所需的时间。

您应该确定修改数据时在查询性能上的提升是否超过了对性能的影响，以及是否需要额外的磁盘空间要求。

唯一索引设计指南

唯一索引能够保证索引键中不包含重复的值，从而使表中的每一行从某种方式上具有唯一性。 只有当唯一性是数据本身的特征时，指定唯一索引才有意义。 例如，如果要确保

`NationalIDNumber` 表中 `HumanResources.Employee` 列的值是唯一的，当主键为 `EmployeeID` 时，对 `NationalIDNumber` 列创建 UNIQUE 约束。如果用户尝试在该列中为多个雇员输入相同的值，将显示错误消息并且不能输入重复的值。

使用多列唯一索引，索引能够保证索引键中值的每个组合都是唯一的。例如，如果为 `LastName`、`FirstName` 和 `MiddleName` 列的组合创建了唯一索引，则表中的任意两行都不会有这些列值的相同组合。

聚集索引和非聚集索引都可以是唯一的。只要列中的数据是唯一的，就可以为同一个表创建一个唯一聚集索引和多个唯一非聚集索引。

唯一索引的优点包括下列几点：

- 能够确保定义的列的数据完整性。
- 提供了对查询优化器有用的附加信息。

创建 PRIMARY KEY 或 UNIQUE 约束会自动为指定的列创建唯一索引。创建 UNIQUE 约束和创建独立于约束的唯一索引没有明显的区别。数据验证的方式是相同的，而且查询优化器不会区分唯一索引是由约束创建的还是手动创建的。但是，如果您的目的是要实现数据完整性，则应为列创建 UNIQUE 或 PRIMARY KEY 约束。这样做才能使索引的目标明确。

注意事项

- 如果数据中存在重复的键值，则不能创建唯一索引、UNIQUE 约束或 PRIMARY KEY 约束。
- 如果数据是唯一的并且您希望强制实现唯一性，则为相同的列组合创建唯一索引而不是非唯一索引可以为查询优化器提供附加信息，从而生成更有效的执行计划。在这种情况下，建议创建唯一索引（最好通过创建 UNIQUE 约束来创建）。
- 唯一非聚集索引可以包括包含性非键列。有关详细信息，请参阅 [具有包含列的索引](#)。

筛选索引设计指南

筛选索引是一种经过优化的非聚集索引，尤其适用于涵盖从定义完善的数据子集中选择数据的查询。筛选索引使用筛选谓词对表中的部分行进行索引。与全表索引相比，设计良好的筛选索引可以提高查询性能、减少索引维护开销并可降低索引存储开销。

适用范围： SQL Server 2008 到 SQL Server 2019 (15.x)。

筛选索引与全表索引相比具有以下优点：

- 提高了查询性能和计划质量

• 定向子集索引能优化计划质量

设计良好的筛选索引可以提高查询性能和执行计划质量，因为它比全表非聚集索引小并且具有经过筛选的统计信息。与全表统计信息相比，经过筛选的统计信息更加准确，因为它们只涵盖筛选索引中的行。

• 减少了索引维护开销

仅在数据操作语言 (DML) 语句对索引中的数据产生影响时，才对索引进行维护。与全表非聚集索引相比，筛选索引减少了索引维护开销，因为它更小并且仅在对索引中的数据产生影响时才进行维护。筛选索引的数量可以非常多，特别是在其中包含很少受影响的数据时。同样，如果筛选索引只包含频繁受影响的数据，则索引大小较小时可以减少更新统计信息的开销。

• 减少了索引存储开销

在没必要创建全表索引时，创建筛选索引可以减少非聚集索引的磁盘存储开销。可以使用多个筛选索引替换一个全表非聚集索引而不会明显增加存储需求。

列中包含查询在 SELECT 语句中引用的定义完善的数据子集时，筛选索引很有用。示例如下：

- 仅包含少量非 NULL 值的稀疏列。
- 包含多种类别的数据的异类列。
- 包含多个范围的值（如美元金额、时间和日期）的列。
- 由列值的简单比较逻辑定义的表分区。

如果索引中的行数与全表索引相比较少时，筛选索引减少的维护开销最为明显。如果筛选索引包含表中的大部分行，则与全表索引相比，其维护开销可能更高。在这种情况下，应使用全表索引而不是筛选索引。

筛选索引是针对一个表定义的，仅支持简单比较运算符。如果需要引用多个表或具有复杂逻辑的筛选表达式，则应创建视图。

设计注意事项

为了设计有效的筛选索引，必须了解应用程序使用哪些查询以及这些查询与您的数据子集有何关联。例如，所含值中大部分为 NULL 的列、含异类类别的值的列以及含不同范围的值的列都属于具有定义完善的子集的数据。以下设计注意事项提供了筛选索引优于全表索引的各种情况。

💡 提示


非聚集索引有缺索引字义支持使用筛选的条件。若两只是减小在 OLTP 表中添加列友

非聚集列存储索引定义又符使用筛选的条件。 若要尽量减少在 OLTP 表中添加列存储索引的性能影响，请使用筛选条件，以便创建仅关于运行工作负荷冷数据的非聚集列存储索引。


数据子集的筛选索引

在列中只有少量相关值需要查询时，可以针对值的子集创建筛选索引。例如，当列中的值大部分为 NULL 并且查询只从非 NULL 值中进行选择时，可以为非 NULL 数据行创建筛选索引。由此得到的索引与对相同键列定义的全表非聚集索引相比，前者更小且维护开销更低。

例如，AdventureWorks2012 数据库中有一个包含了 2679 行的 Production.BillOfMaterials 表。EndDate 列只有 199 行包含非 NULL 值，其他的 2480 行均包含 NULL。下面的筛选索引将涵盖这样的查询：返回在该索引中定义的列的查询，以及只选择 EndDate 中具有非 NULL 值的行的查询。

SQL	 复制
<pre>CREATE NONCLUSTERED INDEX FIBillOfMaterialsWithEndDate ON Production.BillOfMaterials (ComponentID, StartDate) WHERE EndDate IS NOT NULL ; GO</pre>	

筛选索引 FIBillOfMaterialsWithEndDate 对下面的查询有效。您可以显示查询执行计划，以确定查询优化器是否使用了该筛选索引。

SQL	 复制
<pre>SELECT ProductAssemblyID, ComponentID, StartDate FROM Production.BillOfMaterials WHERE EndDate IS NOT NULL AND ComponentID = 5 AND StartDate > '20080101' ;</pre>	

有关如何创建筛选索引以及如何定义筛选索引谓词表达式的详细信息，请参阅 [创建筛选索引](#)。

异类数据的筛选索引

表中含有异类数据行时，可以为一种或多种类别的数据创建筛选索引。

例如，为 Production.Product 表中列出的每种产品均分配了一个 ProductSubcategoryID，后者又与 Bikes、Components、Clothing 或 Accessories 产品类

别相关联。这些类别是异类类别，因为它们在生产表中的列值并不是紧密相关的。例如，对于每种产品类别，列 Color、ReorderPoint、ListPrice、Weight、Class和 Style 均具有唯一特征。假设会经常查询具有子类别 27-36（包含端点）的 Accessories。通过对 Accessories 子类别创建筛选索引，可以提高对 Accessories 的查询的性能，如下例所示。

SQL

复制

```
CREATE NONCLUSTERED INDEX FIPProductAccessories
ON Production.Product (ProductSubcategoryID, ListPrice)
    Include (Name)
WHERE ProductSubcategoryID >= 27 AND ProductSubcategoryID <= 36;
```

筛选索引 FIPProductAccessories 涵盖下面的查询，因为查询

结果包含在该索引中，并且查询计划不包括基表查找。例如，查询谓词表达式 ProductSubcategoryID = 33 是筛选索引谓词 ProductSubcategoryID >= 27 和 ProductSubcategoryID <= 36的子集，查询谓词中的 ProductSubcategoryID 和 ListPrice 列全都是索引中的键列，并且名称作为包含列存储在索引的叶级别。

SQL

复制

```
SELECT Name, ProductSubcategoryID, ListPrice
FROM Production.Product
WHERE ProductSubcategoryID = 33 AND ListPrice > 25.00 ;
```

键列

最好在筛选索引定义中包含少量的键或包含列，并且只包含查询优化器为查询执行计划选择筛选索引所需的列。无论某一筛选索引是否涵盖了查询，查询优化器都可以为查询选择此筛选索引。但是，如果某一筛选索引涵盖了查询，则查询优化器更有可能选择此筛选索引。

在某些情况下，筛选索引涵盖查询，但没有将筛选索引表达式中的列作为键或包含列包括在筛选索引定义中。以下准则说明了筛选索引表达式中的列何时应为筛选索引定义中的键或包含列。这些示例引用了此前创建的筛选索引 FIBillofMaterialsWithEndDate 。

如果筛选索引表达式等效于查询谓词并且查询并未在查询结果中返回筛选索引表达式中的列，则筛选索引表达式中的列不需要作为筛选索引定义中的键或包含列。例如， FIBillofMaterialsWithEndDate 涵盖下面的查询，因为查询谓词等效于筛选表达式，并且

查询结果中未返回 EndDate 。 FIBillofMaterialsWithEndDate 不需要将 EndDate 作为筛选索引定义中的键或包含列。

SQL

复制

```
SELECT ComponentID, StartDate FROM Production.BillOfMaterials
WHERE EndDate IS NOT NULL;
```

如果查询谓词在不与筛选索引表达式等效的比较中使用了筛选索引表达式中的某列，则该列应为筛选索引定义中的键或包含列。例如， `FIBillOfMaterialsWithEndDate` 对下面的查询有效，因为它从筛选索引中选择了行的子集。但是，它不涵盖下面的查询，因为在比较 `EndDate` 中使用了 `EndDate > '20040101'`，此比较不与筛选索引表达式等效。查询处理器在不查找 `EndDate` 值的情况下无法执行此查询。因此， `EndDate` 应为筛选索引定义中的键或包含列。

SQL

复制

```
SELECT ComponentID, StartDate FROM Production.BillOfMaterials
WHERE EndDate > '20040101';
```

如果筛选索引表达式中的某列在查询结果集中，则该列应为筛选索引定义中的键或包含列。例如， `FIBillOfMaterialsWithEndDate` 不涵盖下面的查询，因为它在查询结果中返回了 `EndDate` 列。因此， `EndDate` 应为筛选索引定义中的键或包含列。

SQL

复制

```
SELECT ComponentID, StartDate, EndDate FROM Production.BillOfMaterials
WHERE EndDate IS NOT NULL;
```

表的聚集索引键不需要是筛选索引定义中的键或包含列。聚集索引键自动包含在所有非聚集索引（包括筛选索引）中。

筛选谓词中的数据转换运算符

如果筛选索引结果的筛选索引表达式中指定的比较运算符会导致隐式或显式数据转换，则转换发生在比较运算符的左边时，会出现错误。解决方法是在比较运算符的右边编写包含数据转换运算符（`CAST` 或 `CONVERT`）的筛选索引表达式。

下面的示例创建一个包含多种数据类型的表。

SQL

复制

```
USE AdventureWorks2012;
GO
CREATE TABLE dbo.TestTable (a int, b varbinary(4));
```


在下面的筛选索引定义中，列 `b` 隐式转换为整数数据类型，以便与常量 `1` 进行比较。因为转换发生在筛选谓词中运算符的左边，所以这会生成错误消息 10611。

SQL

复制

```
CREATE NONCLUSTERED INDEX TestTabIndex ON dbo.TestTable(a,b)
WHERE b = 1;
```

解决方法是将右侧的常量转换为与列 `b` 的类型相同的类型，如下例所示：

SQL

复制

```
CREATE INDEX TestTabIndex ON dbo.TestTable(a,b)
WHERE b = CONVERT(varbinary(4), 1);
```

将数据转换从比较运算符的左边移动到右边可能会改变转换的含义。在上例中，将 `CONVERT` 运算符添加到右边时，相应的比较从整数比较更改为 `varbinary` 比较。

列存储索引设计指南

columnstore index 是使用列式数据格式（称为列存储）存储、检索和管理数据的技术。有关详细信息，请参阅[列存储索引概述](#)。

有关详细信息，请参阅[列存储索引 - 新增功能](#)。

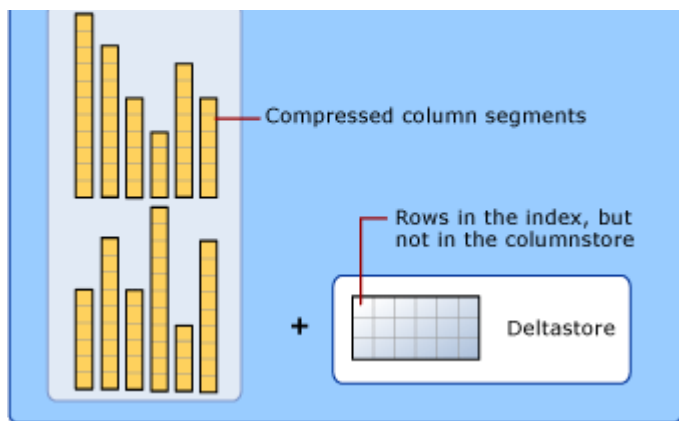
列存储索引体系结构

掌握这些基础知识可以更轻松地理解其他介绍如何有效使用列存储索引的文章。

数据存储使用列存储索引和行存储压缩

在提到列存储索引时，我们使用术语“行存储”和“列存储”来强调数据存储的格式。列存储索引使用这两种类型的存储。





- “列存储”是在逻辑上组织为包含行和列的表、在物理上以按列数据格式存储的数据。

列存储索引使用列存储格式以物理方式存储大部分数据。使用列存储格式时，数据将以列的形式压缩和解压缩。不需要解压缩每个行中未由查询请求的其他值。这样，便可以快速扫描大型表的整个列。

- “行存储”是在逻辑上组织为包含行和列的表、在物理上以按行数据格式存储的数据。这是存储关系表数据（如堆或聚集 B 树索引）的传统方法。

列存储索引还使用称为增量存储的行存储格式以物理方式存储某些行。增量存储（也称为增量行组）是数量太少，不符合压缩到列存储中的条件的行的保存位置。每个增量行组作为聚集 B 树索引实现。

- 增量存储是数量太少，无法压缩到列存储中的行的保存位置。增量存储以行存储格式存储行。

有关列存储术语和概念的详细信息，请参阅[列存储索引：概述](#)。

针对行组和列段执行操作

列存储索引将行分组为可管理的单元。其中每个单元称为一个行组。为提供最佳性能，行组中的行数大到能够提高压缩率，同时又小到能够从内存中操作受益。

例如，列存储索引针对行组执行以下操作：

- 将行组压缩到列存储中。针对行组中的每个列段执行压缩。
- 在 `ALTER INDEX ... REORGANIZE` 操作期间合并行组，包括移除已删除的数据。
- 在 `ALTER INDEX ... REBUILD` 操作期间创建新行组。
- 在动态管理视图 (DMV) 中报告行组运行状况和碎片。

增量存储由一个或多个名为“增量行组”的行组构成。每个增量行组是一个聚集 B 树索引，用于存储较小的大容量加载和插入操作，直到行组包含 1,048,576 行为止，此时，调用 tuple-mover 的进程会自动将关闭的行组压缩到列存储中。

有关行组状态的详细信息，请参阅 [sys.dm_db_column_store_row_group_physical_state](#)。

有关行组优化的详细信息，请参阅 [sys.dm_db_column_store_row_group_physical_stats \(Transact-SQL\)](#)。

💡 提示

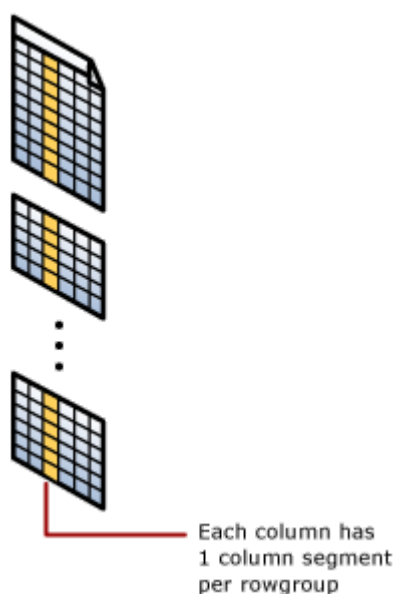
小行组过多会降低列存储索引的质量。重新组织操作将遵循一个内部阈值策略（确定如何移除已删除行并合并已压缩行组）来合并较小的行组。合并后，索引质量应有所提高。

📌 备注

从 SQL Server 2019 (15.x) 开始，tuple-mover 通过后台合并任务获得帮助，该任务会自动压缩较小的已存在一段时间（由内部阈值确定）的 OPEN 增量行组，或者合并已从中删除大量行的 COMPRESSED 行组。

每个列在每个行组中都有自身的一些值。这些值称为“列段”。每个行组包含表中每个列的一个列段。每个列在每个行组中有一个列段。

All Columns



当列存储索引压缩行组时，会单独压缩每个列段。若要解压缩整个列，列存储索引只需解压缩每个行组中的一个列段。

有关列存储术语和概念的详细信息，请参阅[列存储索引：概述](#)。

小规模加载和插入操作转到增量存储

列存储索引一次至少可将 102,400 个行压缩到列存储索引中，以此提高列存储的压缩率和性能。若要批量压缩行，列存储索引可在增量存储中累积小规模加载和插入操作。

增量存储操作在后台处理。若要返回正确的查询结果，聚集列存储索引将合并来自列存储和增量存储的查询结果。

在以下情况下，行将转到增量存储：

- 使用 `INSERT INTO ... VALUES` 语句插入。
- 行位于批量加载操作的末尾，并且编号小于 102,400。
- 更新。每个更新操作实现为删除并插入。

增量存储中还会存储标记为已删除、但实际并未从列存储中删除的已删除行的 ID 列表。

有关列存储术语和概念的详细信息，请参阅[列存储索引：概述](#)。

增量行组已满时将压缩到列存储中

聚集列存储索引最多收集每个增量行组中的 1,048,576 行，达到此数字后，会将行组压缩到列存储中。这可以提高列存储索引的压缩率。在增量行组达到最大行数后，它会从“开启”状态转换为“关闭”状态。名为 tuple-mover 的后台进程会检查已关闭的行组。如果进程找到已关闭的行组，就会压缩行组，并将它存储到列存储中。

压缩增量行组后，现有的增量行组会转换为“逻辑删除”状态，以便稍后由 tuple-mover 在没有引用该行组时将其删除，并将新的压缩行组标记为“COMPRESSED”。

有关行组状态的详细信息，请参阅 [sys.dm_db_column_store_row_group_physical_stats \(Transact-SQL\)](#)。

可以通过使用 `ALTER INDEX` 重新生成或重新组织索引，强制将增量行组压缩到列存储中。请注意，如果在压缩期间遇到内存压力，列存储索引可能会减少压缩行组中的行数。

有关列存储术语和概念的详细信息，请参阅[列存储索引：概述](#)。

每个表分区具有自身的行组和增量行组

索引、堆和列存储索引中的分区概念是相同的。将表分区会根据列值范围将表划分为较小的行组。分区通常用于管理数据。例如，可为每年的数据创建一个分区，然后使用分区切换将数据存档到更经济的存储中。分区切换适用于列存储索引，可让你轻松将数据分区移到另一个位置。

行组始终在表分区中定义。将某个列存储索引分区后，每个分区都有其自身的压缩行组和增量行组。

如果需要从列存储中删除数据，请考虑使用表分区。切出并截断不再需要的分区是一种高效的策略，可以删除数据而不会产生由较小的行组引入的碎片。

每个分区可以包含多个增量行组

每个分区可以包含多个增量行组。如果列存储索引需要将数据添加到增量行组，而增量行组已锁定，则列存储索引会尝试获取其他增量行组中的锁。如果没有任何可用的增量行组，列存储索引将创建新的增量行组。例如，具有 10 个分区的表可以轻松包含 20 个或更多的增量行组。

可以在同一个表中组合列存储索引和行存储索引

非聚集索引包含基础表中部分或全部行与列的副本。索引将定义为表的一个或多个列，并具有一个用于筛选行的可选条件。

从 SQL Server 2016 (13.x) 开始，可以对行存储表创建可更新的非聚集列存储索引。列存储索引将存储数据的副本，因此你需要提供额外的存储。但是，列存储索引中的数据压缩成的大小比行存储表所需的大小更小。如果采取这种做法，你可以同时对列存储索引以及行存储索引上的事务运行分析。当行存储表中的数据更改时，列存储将会更新，因此这两个索引适用于相同的数据。

从 SQL Server 2016 (13.x) 开始，可以对一个列存储索引使用一个或多个非聚集行存储索引。这样，便可以针对基础列存储上执行有效的表查找。其他选项也可供使用。例如，可以通过在行存储表中使用 UNIQUE 约束来强制主键约束。由于不唯一的值无法插入行存储表，SQL Server 无法将值插入列存储。

性能注意事项

- 非聚集列存储索引定义支持使用筛选的条件。若要尽量减少在 OLTP 表中添加列存储索引的性能影响，请使用筛选条件，以便创建仅关于运行工作负荷冷数据的非聚集列存储索引。
- 一个内存中表可以有一个列存储索引。你可以在创建表时创建它，也可以稍后使用 [ALTER TABLE \(Transact-SQL\)](#) 来添加。在低于 SQL Server 2016 (13.x) 的版本中，仅基于磁盘的表可以有列存储索引。

有关详细信息，请参阅[列存储索引 - 查询性能](#)。

设计指南

- 一个行存储表可以有一个可更新的非聚集列存储索引。在低于 SQL Server 2014 (12.x) 的版本中，非聚集列存储索引是只读的。

有关详细信息，请参阅[列存储索引 - 设计指南](#)。

哈希索引设计指南

所有内存优化表都至少必须有一个索引，因为行正是通过索引才连接在一起。在内存优化表中，每个索引也经过内存优化。哈希索引是内存优化表中可能存在的索引类型之一。有关详细信息，请参阅[内存优化表的索引](#)。

适用范围： SQL Server 2014 (12.x) 到 SQL Server 2019 (15.x)。

哈希索引体系结构

哈希索引包含一个指针数组，该数组的每个元素被称为哈希桶。

- 每个桶为 8 个字节，用于存储键项的链接列表的内存地址。
- 每个条目是索引键的值，以及其在基础内存优化表中的对应行的地址。
- 每个条目指向条目的链接列表中的下一个条目，所有都链接到当前桶。

必须在定义索引时指定哈希桶的数量：

- 桶与表行或非重复值的比例越低，桶链接列表的平均长度就越长。
- 较短的链接列表比较长的链接列表执行更快。
- 哈希索引中的最大哈希桶数目为 1,073,741,824。

提示

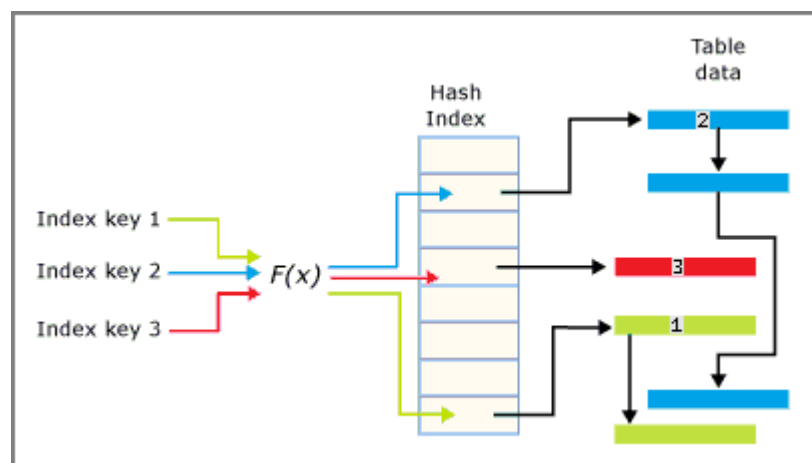
若要为你的数据确定合适的 `BUCKET_COUNT`，请参阅 [配置哈希索引桶计数](#)。

哈希函数适用于索引键列，其结果用于确定键位于哪个哈希桶中。每个哈希桶都包含一个指向行的指针，这些行的哈希键值映射到该哈希桶。

用于哈希索引的哈希函数具有以下特征：

- SQL Server 拥有一个用于所有哈希索引的哈希函数。
- 哈希函数具有确定性。相同的输入键值始终映射到哈希索引中的同一哈希桶。
- 多个索引键可能映射到同一个哈希 Bucket。
- 哈希函数经过均衡处理，这意味着索引键值在哈希桶上的分布通常符合泊松分布或钟型曲线分布，而不是平坦的线性分布。
- 泊松分布并非均匀分布。索引键值并非均匀地分布在哈希 Bucket 中。
- 如果两个索引键映射到同一哈希桶，则产生哈希冲突。大量哈希冲突可影响读取操作的性能。现实的目标是 30% 的桶包含两个不同的键值。

下图汇总了哈希索引和桶的交互作用。



配置哈希索引桶计数

哈希索引桶计数在索引创建时指定，可使用 `ALTER TABLE...ALTER INDEX REBUILD` 语法进行更改。

在大多数情况下，桶计数在理想情况下应该介于索引键中非重复值数目的 1 到 2 倍之间。

可能无法始终预测到某个特定索引键可能具有或将具有多少个值。如果 `BUCKET_COUNT` 值在键值的实际数目的 10 倍以内，性能表现通常依然良好，并且高估通常比低估要好。

桶 **太少** 会带来以下缺点：

- 增多非重复键值的哈希冲突。
- 每个非重复值被迫与其他非重复值共享同一个桶。
- 每个桶的平均链长度将会增加。
- 桶链的长度越长，在索引中进行相等性查找的速度就越慢。

桶 **太多** 会带来以下缺点：

- 桶计数过高可能会导致空桶增多。
- 空桶会影响全文检索扫描的性能。如果定期执行这些操作，请考虑选择接近非重复索引键值数目的桶计数。
- 空桶会使用内存，尽管每个桶只用 8 个字节。

① 备注

添加更多的桶无益于减少共享重复值的、链接在一起的条目。值重复率用于确定哈希是否为适当的索引类型，而不是用于计算桶计数。

性能注意事项

哈希索引的性能为：

- 当 `WHERE` 子句中的谓词为哈希索引键的每一列指定确切值时表现极好。如果有不等谓词，则哈希索引将恢复为扫描。
- 当 `WHERE` 子句中的谓词查找索引键中的一系列值时表现不佳。
- 当 `WHERE` 子句中的谓词为双列哈希索引键的第一列规定了一个特定值，但没有为该键的其他列指定值时表现不佳。

💡 提示

谓词必须包括哈希索引键中的所有列。哈希索引需要键（哈希）才能仔细查找索引。如果索引键由两列组成，但 `WHERE` 子句仅提供第一列，则 SQL Server 没有完整的键可进行哈希运算。这将产生索引扫描查询计划。

如果使用哈希索引并且唯一索引键的数目是行计数的 100 倍（或者更多），则考虑增加桶计数以避免较大的行链，或改用[非聚集索引](#)。


声明注意事项

哈希索引只能存在于内存优化表中，而不能存在于基于磁盘的表中。

可以将哈希索引声明为：

- “唯一”，或可以默认为“非唯一”。
- `NONCLUSTERED` 为默认值。

以下是在 `CREATE TABLE` 语句之外创建哈希索引的语法示例：

SQL	 复制
<pre>ALTER TABLE MyTable_memop ADD INDEX ix_hash_Column2 UNIQUE HASH (Column2) WITH (BUCKET_COUNT = 64);</pre>	

行版本和垃圾回收

在内存优化表中，如果某行受 `UPDATE` 影响，表将创建该行的更新版本。在更新事务期间，其他会话也许能够读取较旧版本的行，从而避免与行锁相关的性能下降。

哈希索引可能也会提供不同的条目版本来适应更新。

以后，当不再需要旧版本时，垃圾回收 (GC) 线程将遍历桶及其链接列表，以清理旧条目。如果链接列表链长度较短，GC 线程的执行效果会更佳。有关详细信息，请参阅[内存中 OLTP 垃圾回收](#)。

内存优化非聚集索引设计指南

非聚集索引是内存优化表中可能存在的一种索引类型。有关详细信息，请参阅[内存优化表的索引](#)。

适用范围： SQL Server 2014 (12.x) 到 SQL Server 2019 (15.x)。

内存中非聚集索引体系结构

内存中非聚集索引最初由 Microsoft Research 在 2011 年提出设想并说明，并使用称为 Bw 树的数据结构实现。Bw 树是 B 树的无锁和无门锁定变体。有关详细信息，请参阅[Bw 树：新硬件平台的 B 树](#)。

Bw 树处于一个非常高的级别，可以理解为按页 ID (PidMap) 组织的页映射，用于分配和重复使用页 ID (PidAlloc) 的设施，和在页映射中链接并相互链接的一组页。这三个高级别子组件组成了 Bw 树的基本内部结构。

该结构一定程度上类似于常规 B 树，每页都有一组经过排序的键值，索引中的每个级别都指向更低的级别，并且叶级别指向数据行。但也存在一些差异。

与哈希索引类似，多个数据行可链接在一起（版本）。级别之间的页指针是逻辑页 ID，这些逻辑页 ID 是页映射表中的偏移量，该表又具有每页的物理地址。

索引页没有就地更新。因此引入了新的增量页。

- 页更新不需要门锁定。
- 索引页没有固定大小。

每个非叶级别页中描述的键值是其指向的子级所包含的最高值，每一行还包含了该页逻辑页 ID。在叶级别页上，除键值外，还包含数据行的物理地址。

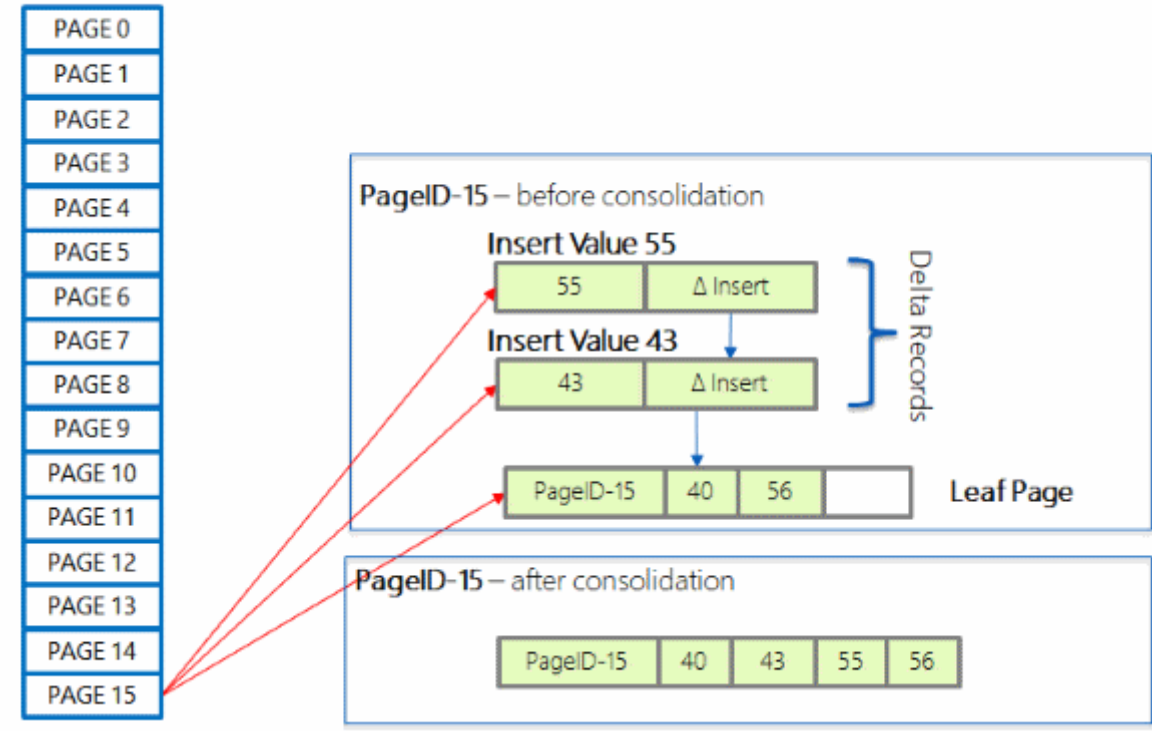
点查找与 B 树相似，但由于仅在一个方向链接页，SQL Server 数据库引擎跟随右页指针，其中每个非叶页具有其子级的最高值，而不是 B 树中的子级最低值。

必须更改叶级别页时，SQL Server 数据库引擎不修改该页本身。相反，SQL Server 数据库引擎创建描述更改的增量记录，并将其追加到先前的页。然后将先前页的页映射表地址更新为增量记录的地址，使该地址成为当前页的物理地址。

增量整合

增量记录的长链最终可能会降低搜索性能，因为这意味着通过索引搜索时，需要遍历这些长链。如果向已有 16 个元素的链添加了新的增量记录，该增量记录中的更改将整合到引用的索引页中，然后将重新生成该页，其中包括触发整合的新增量记录指示的更改。重新生成的新页使用相同的页 ID，但使用新的内存地址。

Page Mapping Table

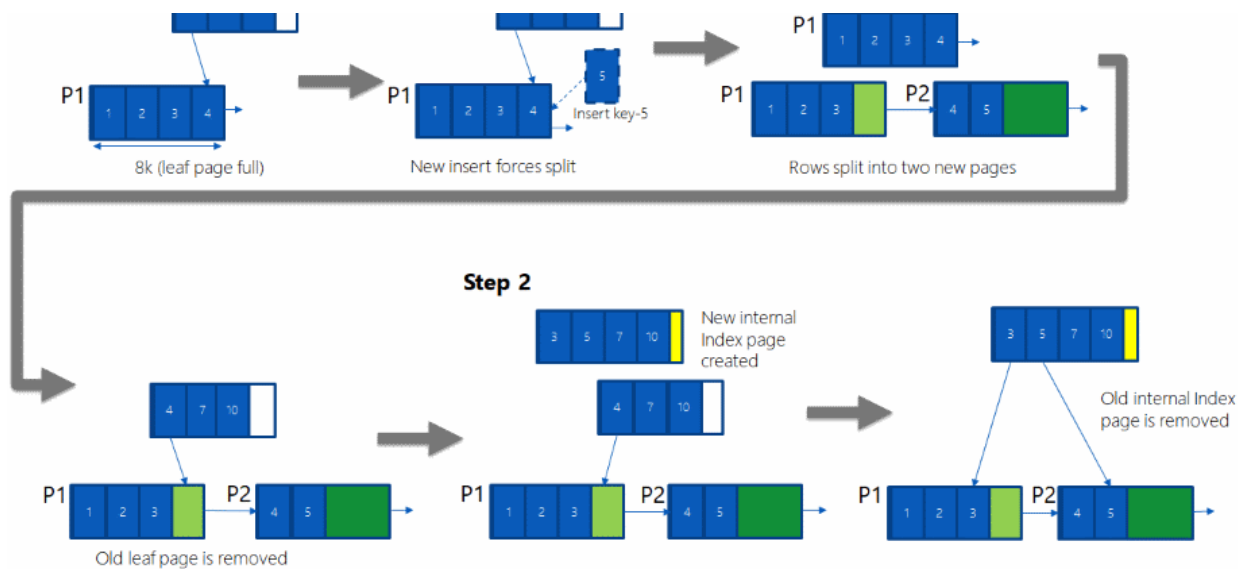


拆分页

Bw 树中的索引页可按需增大，从存储单一行的开始大小开始，最多可存储 8 KB 的索引页。索引页增大到 8 KB 后，插入一个新行会导致拆分索引页。对于内部页，这表示不再有添加另一个键值和指针的空间，对于叶页，这表示所有增量记录整合在一起后，行会因为太大而不能容纳在页中。叶页的页标头中的统计信息会持续跟踪整合增量记录所需的空空间，该信息还会随着每个新增量记录的添加不断调整。

拆分操作通过两个原子步骤完成。在下图中，假设因为正在插入值为 5 的键强制进行了叶页拆分，并存在一个指向当前叶级别页（键值 4）末尾的非叶页。





步骤 1： 分配两个新页（P1 和 P2），将旧 P1 页中的行拆分到这些新页上（包括新插入的行）。使用页映射表中的新槽存储 P2 页的物理地址。此时，任何并发操作都还无法访问 P1 和 P2 页。此外，设置了从 P1 指向 P2 的逻辑指针。然后，在一个原子步骤中更新页映射表，将指针从旧 P1 更改到新 P1。

步骤 2： 非叶页指向 P1，但是没有指针从非叶页直接指向 P2。只能通过 P1 到达 P2。要创建从非叶页指向 P2 的指针，需要分配新的非叶页（内部索引页），复制旧的非叶页中的所有行，并添加一个指向 P2 的新行。完成此操作后，在一个原子步骤中更新页映射表，将指针从旧的非叶页更改为新的非叶页。

合并页

如果 **DELETE** 操作导致某页的大小低于最大页大小（当前为 8 KB）的 10%，或该页上只有一行，那么该页会与相邻的页合并。

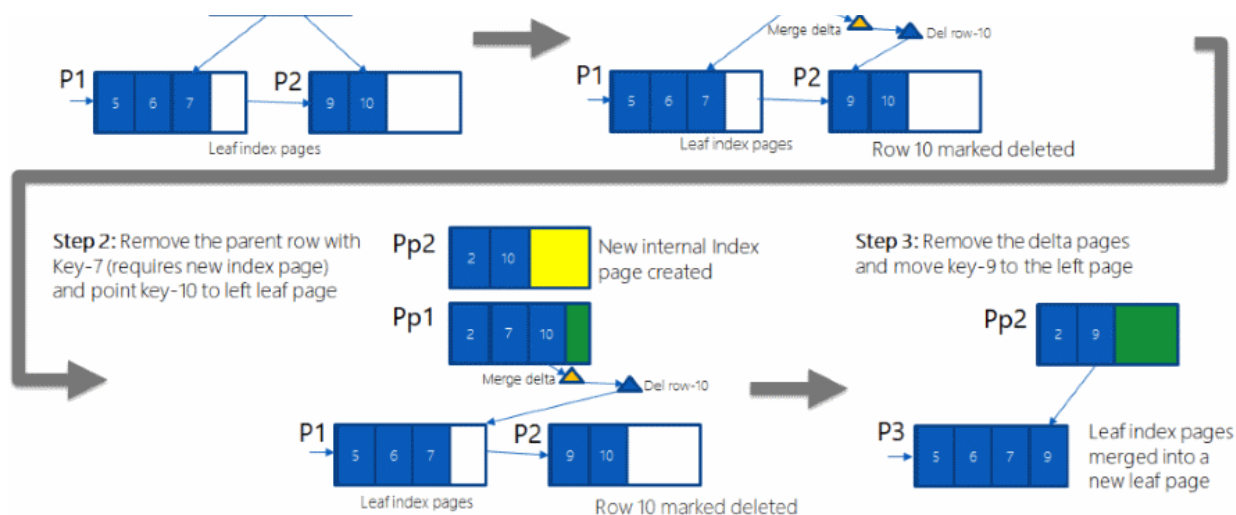
从某页中删除行时，会添加该删除操作的增量记录。此外，还会进行相关检查，确定索引页（非叶页）是否符合合并条件。此检查还会验证删除行之后，剩余空间是否小于最大页大小的 10%。如果符合合并条件，可通过三个原子步骤完成合并。

下图中，假设 **DELETE** 操作删除键值 10。

Operation: Delete row 10



Step 1: Insert delta page for merge and for row-10



步骤 1: 创建表示键值 10（蓝色三角形）的增量页，并将该增量页在非叶页 Pp1 上的指针设置为新的增量页。此外，创建一个特殊的合并增量页（绿色三角形），并链接该页，使其指向增量页。在此阶段，这两个页（增量页和合并增量页）对任何并发事务都不可见。在一个原子步骤中，页映射表中指向叶级别页 P1 的指针更新为指向合并增量页的指针。执行此步骤之后，Pp1 中键值 10 对应的项现在指向合并增量页。

步骤 2: 需要删除非叶页 Pp1 中表示键值 7 的行，然后将键值 10 对应的项更新为指向 P1。为此，需要分配新的非叶页 Pp2，复制 Pp1 中除表示键值 7 的行之外的所有行，然后将键值 10 表示的行更新为指向页 P1。完成此操作后，在一个原子步骤中将 Pp1 的页映射表入口点更新为指向 Pp2。无法再访问 Pp1。

步骤 3: 合并叶级别页 P2 和 P1，并删除增量页。为此，分配新页 P3，合并 P2 和 P1 中的行，并在新的 P3 中包含增量页的更改。然后，在一个原子步骤中将页 P1 的页映射表入口点更新为指向页 P3。

性能注意事项

使用不等谓词查询内存优化表时，非聚集索引的性能优于非聚集哈希索引。

❗ 备注

内存优化的表中的列可以同时为哈希索引和非聚集索引的一部分。

💡 提示

非聚集索引键列中的列包含许多重复值时，更新、插入和删除的性能会降低。在这种情况下提高性能的一种方法是向非聚集索引添加另一列。

其他阅读主题

[CREATE INDEX \(Transact-SQL\)](#)
[ALTER INDEX \(Transact-SQL\)](#)
[CREATE XML INDEX \(Transact-SQL\)](#)
[CREATE SPATIAL INDEX \(Transact-SQL\)](#)
[重新组织和重新生成索引](#)
[使用 SQL Server 2008 索引视图提高性能](#)
[Partitioned Tables and Indexes](#)
[创建主键](#)
[内存优化表的索引](#)
[列存储索引概述](#)
[内存优化表的哈希索引疑难解答](#)
[内存优化表动态管理视图 \(Transact-SQL\)](#)
[与索引相关的动态管理视图和函数 \(Transact-SQL\)](#)
[计算列上的索引](#)
[索引和 ALTER TABLE](#)
[自适应索引碎片整理](#) 

此页面有帮助吗？

 是  否
