别再一知半解了，索引其实就那么回事~

CodeSheep 2022-02-25 14:56

The following article is from 业余码农 Author Amazing10

业余码农

抖音全干开发工程师，分享计算机基础、面试技巧、职场进阶思维。

往期干货笔记整理

- 熬夜肝了个Linux速查备忘手册.pdf
- 我的浏览器收藏夹大公开
- 数据结构和算法刷题笔记.pdf下载
- LeetCode算法刷题C/C++版答案pdf下载
- LeetCode算法刷题Java版答案pdf下载
- 找工作简历模板集(word格式)下载
- Java基础核心知识大总结.pdf 下载

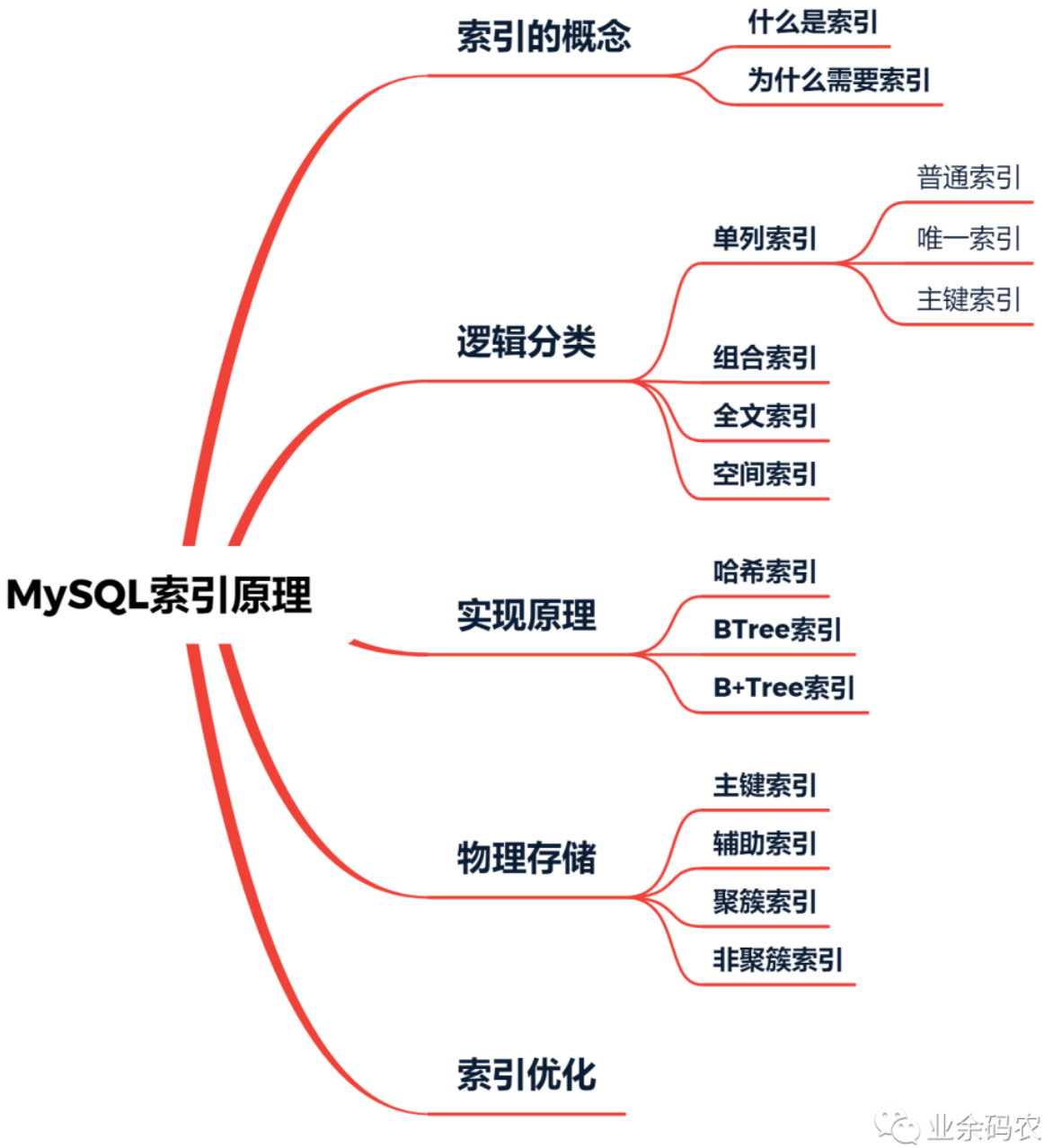
- **C/C++常见面试题（含答案）下载**
- **设计模式学习笔记.pdf下载**
- **Java后端开发学习路线+知识点总结**
- **前端开发学习路线+知识点总结**
- **大数据开发学习路线+知识点总结**
- **C/C++(后台)学习路线+知识点总结**
- **嵌入式开发学习路线+知识点总结**

索引的概念基本所有人都会遇到过，就算没有了解过数据库中的索引，在生活中也不可避免的接触到。比方说书籍的目录，字典的查询页，图书馆的科目检索等等。其实这些都是一种索引，并且所起到的作用大同小异。

而对于数据库而言，只不过是将索引的概念抽象出来，让建立索引的过程更为灵活而自由，从而可以在不同的场景下优化数据库的查询效率。

索引在数据库的实际应用场景中十分普遍，数据库的优化也离不开对索引的优化。同时，索引相关的知识也是面试高频的考点之一，是应试者理论结合现实最为直接的体现。

因此，本文我们将从基础理论出发，介绍 MySQL 按照逻辑角度的索引分类和实现，通过数据结构的实现原理阐述不同结构对建立索引带来的优劣势，同时针对物理存储的方式对索引的组织特点和应用场景进行分析。最后根据不同的应用场景尽可能的探究如何建立起高性能的索引。文章结构如下：



1 概念

什么是索引

索引似乎并没有十分明确的定义，更多的是一种定性的描述。简单来讲，索引就是一种将数据库中的记录按照特殊形式存储的数据结构。通过索引，能够显著地提高数据查询的效率，从而提升服务器的性能。

专业一点来说呢，索引是一个排好序的列表，在这个列表中存储着索引的值和包含这个值的数据所在行的物理地址。在数据库十分庞大的时候，索引可以大大加快查询的速度，这是因为使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。

说起索引，其实并不是 MySQL 数据库特有的机制，在关系型数据库中都会有类似不同的实现。这里我们也只是讨论 MySQL 数据库中的索引实现。

事实上，说是 MySQL 的索引其实并不准确。因为在 MySQL 中，索引是在存储引擎层而不是服务器层实现的。这意味着我们所讨论的索引准确来说是 InnoDB 引擎或 MyISAM 引擎或其它存储引擎所实现的。

所以索引即便是在 MySQL 中也没有统一的标准，不同存储引擎的所实现的索引工作方式也并不一样。不是所有的存储引擎都支持相同类型的索引，即便是多个引擎支持同一种类型的索引，其底层的实现也可能不同。

为什么需要索引

说了这么多，索引似乎就是给数据库添加了一个「目录页」，能够方便查询数据。但是索引的作用就仅此而已了吗，为什么需要大费周章的建立并优化索引？

说个题外话，我其实查字典从来都不喜欢查目录页，无论是查中文还是英文。因为觉得那样很慢，一个个找索引，效率很低。我习惯用的方式就是直接翻开字典，根据翻开的位置进行前后调整。比方说我想找「酱 JIANG」字，会先随机翻到一页，可能是「F」开头，在「J」前面，就往后翻一点；如果随机翻到「L」，那就往前翻一点。重复直至找到。

这大概就是类似于二分查找的方式，看起来好像是摆脱了索引的束缚，并且也能够获得比较高的查询效率。但是其实转念一想，在计算机的运行处理中，「一个个找索引」这个过程其实非常快，不能跟我们手动比对偏旁部首的效率相提并论。同时，为什么我可以直接翻开字典根据字母进行调整呢，这其实不就是因为我的脑子里存在一个大概的「索引表」，知道每个字母大概对应于字典的哪一个位置。虽然是模糊的，但却是真实存在的。（好不容易强行解释了一波...）

如此一来，可以看出索引的一大好处是如其概念中所提及的，使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。这样的方式自然减少了服务器在响应时所需要对数据库扫描的数据量。

不仅如此，在执行数据库的范围查询时，若不使用索引，那么MySQL会先扫描数据库的所有行数据并从中筛选出目标范围内的行记录，将这些行记录进行排序并生成一张临时表，然后通过临时表返回用户查询的目标行记录。这个过程会涉及到临时表的建立和行记录的排序，当目标行记录较多的时候，会大大影响范围查询的效率。

所以当添加索引时，由于索引本身具有的顺序性，使得在进行范围查询时，所筛选出的行记录已经排好序，从而避免了再次排序和需要建立临时表的问题。

同时，由于索引底层实现的有序性，使得在进行数据查询时，能够避免在磁盘不同扇区的随机寻址。使用索引后能够通过磁盘预读使得在磁盘上对数据的访问大致呈顺序的寻址。这本质上

是依据局部性原理所实现的。

> 局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。由于磁盘顺序读取的效率很高(不需要寻道时间，只需很少的旋转时间)，因此对于具有局部性的程序来说，磁盘预读可以提高I/O效率。

磁盘预读要求每次都会预读的长度一般为页的整数倍。而且数据库系统将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。这里的页是通过页式的内存管理所实现的，概念在这里简单提一嘴。

> 分页机制就是把内存地址空间分为若干个很小的固定大小的页，每一页的大小由内存决定。这样做是为了从虚拟地址映射到物理地址，提高内存和磁盘的利用率。

所以呢，总结一下。索引的存在具有很大的优势，主要表现为以下三点：

> - 索引大大减少了服务器需要扫描的数据量
> - 索引可以帮助服务器避免排序和临时表
> - 索引可以将随机 I/O 变成顺序 I/O

以上三点能够大大提高数据库查询的效率，优化服务器的性能。因此一般来说，为数据库添加高效的索引对数据库进行优化的重要工作之一。

不过，凡事都有两面性。索引的存在能够带来性能的提升，自然在其它方面也会付出额外的代价。

> 索引本身以表的形式存储，因此会占用额外的存储空间；
>
> 索引表的创建和维护需要时间成本，这个成本随着数据量增大而增大；
>
> 构建索引会降低数据的修改操作（删除，添加，修改）的效率，因为在修改数据表的同时还需要修改索引表；

所以对于非常小的表而言，使用索引的代价会大于直接进行全表扫描，这时候就并不一定非得使用索引了。没办法，成年人的世界总是这么的趋利避害。

2 逻辑分类

从逻辑的角度来对索引进行划分的话，可以分为单列索引、全文索引、组合索引和空间索引。其中单列索引又可分为主键索引、唯一索引和普通索引。这里的逻辑可以理解为从 SQL 语句的角度，或者是从数据库关系表的角度。下面就简单介绍这些索引的作用和用法，以及在修改表的时候如何添加索引。

主键索引

即主索引，根据主键建立索引，不允许重复，不允许空值；

> 主键：数据库表中一列或列组合（字段）的值，可唯一标识表中的每一行。

加速查询 + 列值唯一（不可以有**null**）+ 表中只有一个

```
ALTER TABLE 'table_name' ADD PRIMARY KEY pk_index('col');
```

唯一索引

用来建立索引的列的值必须是唯一的，允许空值。唯一索引不允许表中任何两行具有相同的索引值。比方说，在 `employee` 表中职员的姓 `name` 上创建了唯一索引，那么就表示任何两个员工都不能同姓。

加速查询 + 列值唯一（可以有**null**）

```
ALTER TABLE 'table_name' ADD UNIQUE index_name('col');
```

普通索引

用表中的普通列构建的索引，没有任何限制。

仅加速查询

```
ALTER TABLE 'table_name' ADD INDEX index_name('col');
```

全文索引

用大文本对象的列构建的索引

```
ALTER TABLE 'table_name' ADD FULLTEXT INDEX ft_index('col');
```

组合索引

用多个列组合构建的索引，这多个列中的值不允许有空值。

多列值组成一个索引，专门用于组合搜索，其效率大于索引合并。

```
ALTER TABLE 'table_name' ADD INDEX index_name('col1','col2','col3');
```

在对多列组合建立索引时，会遵循「最左前缀」原则。

> 最左前缀原则：顾名思义，就是最左优先，上例中我们创建了 (col1, col2, col3) 多列索引,相当于创建了 (col1) 单列索引，(col1, col2) 组合索引以及 (col1, col2, col3) 组合索引。

所以当我们在创建多列索引时，要根据业务场景，将 where 子句中使用最频繁的一列放在最左边。

空间索引

对空间数据类型的字段建立的索引，底层可通过 R 树实现。只不过使用较少，了解即可。

3 实现原理

我们知道，索引的底层本身就是通过数据结构来进行实现的。那么根据其底层的结构，常见的索引类型可分为哈希索引，BTree 索引，B+Tree 索引等。这里我们就主要来介绍这三种索引背后的实现机制。

哈希索引

顾名思义，哈希索引是通过哈希表实现的。哈希表的特点在之前的文章「九大数据结构」中已经详细介绍过。通过哈希表的键值之间的对应关系，能够在查询时精确匹配索引的所有列。哈希索引将所有的根据索引列计算出来的哈希码存储在索引中，同时将指向每个数据行的指针保存在哈希表中。

上图是通过哈希索引查询行数据的示意图，可以发现哈希索引同样会发生哈希冲突，并且是通过链地址法解决冲突的。当发送冲突时，还需要对链表进行遍历对比，才能够找到最终的结果。

在 MySQL 中，只有 Memory 存储引擎显式的支持哈希索引，而innodb是隐式支持哈希索引的。

> 这里的隐式支持是指，innodb引擎有一个特殊的功能“自适应哈希索引”，当innodb注意到一些索引值被使用的非常频繁时，且符合哈希特点（如每次查询的列都一样），它会在内存中基于 B-Tree 索引之上再创建一个哈希索引。这样就让 BTree 索引也具有哈希索引的一些有点。这是一个完全自动的、内部的行为。

由于哈希结构的特殊性，其用于非常高的检索效率，通过哈希函数的映射可以一步到位。但是同样也是因为结构的特殊，导致哈希索引只适用于某些特定的场合。哈希索引的限制[1]：

1. 不支持范围查询，比如 `WHERE a > 5` ；只支持等值比较查询，包括 `=` 、 `IN` 、 `<=>`
2. 无法被用来避免数据的排序操作；因为经过了哈希函数的映射过程，使得丢失了哈希前后的大小关系，从而无法按照索引值的顺序存储。
3. 不支持部分索引列的匹配查找，因为哈希索引始终使用索引列的全部内容来计算哈希值。例如在数据列 (A, B) 上建立哈希索引，如果查询只有数据列 A，则无法使用该索引。
4. 无法避免表扫描。因为当出现哈希冲突的时候，存储引擎必须遍历链表（拉链法）中所有的行指针，逐行进行比较，直到找到所有符合条件的行。
5. 哈希冲突很多的情况下，其索引维护的代价很高，并且性能并不一定会比 BTree 索引高。

BTree 索引

BTree 实际上是一颗多叉平衡搜索树。从名字可以看出，BTree 首先是一颗多叉搜索树，这意味着它是具有顺序的；其次 BTree 还是平衡的，这意味着它的左右子树高度差小于等于1。

事实上一颗 BTree 需要满足以下几个条件：

每个叶子结点的高度都是一样的；

每个非叶子结点由 n-1 个 key 和 n 个指针 point 组成，其中 d<=n<=2d, key 和 point 相互间隔，结点两端一定是 key；

叶子结点指针都为 null；

非叶子结点的key都是 [key, data] 二元组，其中 key 表示作为索引的键，data 为键值所在行的数据；

一颗常见的BTree树见下图。

这是一颗三阶的BTree，可通过键值的大小排序进行数据的查询和检索，其中叶子节点的指针都为空，因此省略没画。从上图可以发现，BTree 的树形状相较于我们之前常见的二叉树等结构，更为扁平和矮胖。

之所以这样设计，还是跟磁盘读取的特点有关。我们知道在建立索引时，也是需要占据物理空间的。而实际上当数据量比较大的时候，索引文件的大小也十分吓人。考虑到一个表上可能有多个索引、组合索引、数据行占用更小等情况，索引文件的大小可能达到物理盘中数据的1/10，甚至可达到1/3。

这就意味着索引无法全部装入内存之中。当通过索引对数据进行访问时，不可避免的需要对磁盘进行读写访问。同时我们还知道，内存的读写速度是磁盘的几个数量级。因此在对索引结构进行设计时要尽可能的减少对磁盘的读写次数，也就是所谓的磁盘 I/O 次数。

这也就是索引会采用 BTree 这种扁平树结构的原因，树的层数越少，磁盘I/O的次数自然就越少。不仅如此，我们上面提到过磁盘预读的局部性原理。根据这个原理再加上页表机制，能够在进行磁盘读取的时候更大化的提升性能。

BTree 相较于其它的二叉树结构，对磁盘的 I/O 次数已经非常少了。但是在实际的数据库应用中仍有些问题无法解决。

一是无法定位到数据行。通过 BTree 可以根据主键的排序定位出主键的位置，但是由于数据表的记录有多个字段，仅仅定位到主键是不够，还需要定位到数据行。虽然这个问题可以通过在 BTree 的节点中存储数据行或者增加定位的字段，但是这种方式会使得 BTree 的深度大幅度提高，从而也导致 I/O 次数的提高。

二是无法处理范围查询。在实际的应用中，数据库范围查询的频率非常高，而 BTree 只能定位到一个索引位置。虽然可以通过先后查询范围的左右界获得，但是这样的操作实际上无法很好的利用磁盘预读的局部性原理，先后查询可能会造成通过预读取的物理地址离散，使得 I/O 的效率并不高。

三是当数据量一大时，**BTree**的高度依旧会变得很高，搜索效率还是会大幅度的下降。

问题总是推动改进的前提。基于以上的问题考虑，就出现了对 BTree 的优化版本，也就是B+Tree。

B+Tree索引

B+Tree 一看就是在 BTree 的基础上做了改进，那么到底改变了什么呢。废话不多说，先上图。

上图实际上是一种带有顺序索引的 B+Tree，与最基本的 B+Tree 的区别就在于叶子节点是否通过指针相连。一般数据库中常用的结构都是这种带有顺序索引的 B+Tree。后文探讨的也都是带顺序索引的 B+Tree 结构。

对比 BTree 和 B+Tree，我们可以发现二者主要在以下三个方面上的不同：

> 1. 非叶子节点只存储键值信息，不再存储数据。
> 2. 所有叶子节点之间都有一个链指针，指向下一个叶子节点。
> 3. 数据都存放在叶子节点中。

看着 B+Tree，像不像是一颗树与一个有序链表的结合体。因而其实 B+Tree 也就是带有树和链表的部分优势。树结构使得有序检索更为简单，I/O 次数更少；有序链表结构使得可以按照键值排序的次序遍历全部记录。

B+Tree 在作为索引结构时能够带来的好处有：

一，**I/O** 次数更少。这是因为上文也说过，BTree 的节点是存放在内存页中的。那么在相同的内存页大小（一般为4k）的情况下，B+Tree 能够存储更多的键值，那么整体树结构的高度就会更小，需要的 I/O 次数也就越小。

二，数据遍历更为方便。这个优势很明显是由有序链表带来的。通过叶子节点的链接，使得对所有数据的遍历只需要在线性的链表上完成，这就非常适合区间检索和范围查询。

三，查询性能更稳定。这自然是由于只在叶子节点存储数据，所以所有数据的查询都会到达叶子节点，同时叶子节点的高度都相同，因此理论上来说所有数据的查询速度都是一致的。

正是由于 B+Tree 优秀的结构特性，使得常用作索引的实现结构。在 MySQL 中，存储引擎 MyISAM 和 InnoDB 都分别以 B+Tree 实现了响应的索引设计。

4 物理存储

虽说 B+Tree 结构都可以用在 MyISAM 和 InnoDB，但是这二者对索引的在物理存储层次的实现方式却不相同。InnoDB 实现的是聚簇索引，而 MyISAM 实现的却是非聚簇索引。在介绍聚簇索引之前，我们需要先了解以下啥是佩奇，不对，是啥是「主键索引」和「辅助索引」。

其实概念很简单。我们刚刚不是在讲 B+Tree 的时候说过，树的非叶子节点只存储键值。没错就是这个键值，当这个键值是数据表的主键时，那么所建立的就是主键索引；当这个键值是其它字段的时候，就是辅助索引。因而可以得出，主键索引只能有一个，而辅助索引却可以有很多个。

聚簇索引和非聚簇索引的区别也就是根据其对应的主键索引和辅助索引的不同特点而实现的。

聚簇索引

说回聚簇索引。先丢个定义。

> 聚簇索引的主键索引的叶子结点存储的是键值对应的数据本身；辅助索引的叶子结点存储的是键值对应的数据的主键键值。

这句话的信息量挺大的。首先，分析第一句话，主键索引的叶子节点存储的是键值对应的数据本身。

我们知道，主键索引存储的键值就是主键。那么也就是说，聚簇索引的主键索引，在叶子节点中存储的是主键和主键对应的数据。数据和主键索引是存储在一起的，一起作为叶子节点的一部分。

然后，分析第二句话，辅助索引的叶子结点存储的是键值对应的数据的主键键值。

我们又知道，辅助索引存储的键值是非主键的字段。那就也就是说，通过辅助索引，可以找到非主键字段对应的数据行中的主键。

重点来了。当然主键索引和辅助索引一结合，能干啥呢。当直接采用主键进行检索时，可通过主键索引直接获得数据；而当采用非主键进行检索时，先需要通过辅助索引来获得主键，然后再通过这个主键在主键索引中找到对应的数据行。

举个例子吧。假设有这么一个数据表。

那么采用聚簇索引的存储方式，对应的主键索引为：（主键为ID）

对应的辅助索引为：（键值为Name，大概的意思）：

所以当使用 `where ID = 7` 这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。第二步使用主键在主键索引B+树种再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。

最后把以上过程整理总结一下，聚簇索引实际上的过程就分为以下两个过程。现在这个图应该能够看懂了吧。

非聚簇索引

学完了聚簇索引，非聚簇索引就简单多啦。同样，先上定义。

> 非聚簇索引的主键索引和辅助索引几乎是一样的，只是主索引不允许重复，不允许空值，他们的叶子结点都存储指向键值对应的数据的物理地址。

与聚簇索引来对比着看，上面的定义能够说明什么呢。首先，主键索引和辅助索引的叶子结点都存储着键值对应的数据的物理地址，这说明无论是主键索引还是辅助索引都能够通过直接获得数据，而不需要像聚簇索引那样在检索辅助索引时还得多绕一圈。

同时还说明一个点，叶子结点存储的是物理地址，那么表示数据实际上是存在另一个地方的，并不是存储在B+树的结点中。这说明非聚簇索引的数据表和索引表是分开存储的。

同样，对非聚簇索引的检索过程来个总结。

无论是主键索引还是辅助索引的检索过程，都只需要通过相应的 B+Tree 进行搜索即可获得数据对应的物理地址，然后经过依次磁盘 I/O 就可访问数据。

对比聚簇索引和非聚簇索引，可以发现二者最主要的区别就是在于是否在 B+Tree 的节点中存储数据，也就是数据和索引是否存储在一起。这个区别导致最大的问题就是聚簇索引的索引的顺序和数据本身的顺序是相同的，而非聚簇索引的顺序跟数据的顺序没有啥关系。

5 索引优化

介绍了这么多的索引，其实最终都是为了建立高性能的索引策略，对数据库中的索引进行优化。索引的优化有很多角度，针对特定的业务场景可采用不同的优化策略。这里考虑到文章篇幅，就不具体介绍，下次再出一篇专门讲索引优化的文章。简单列举一下在进行优化时可以考虑的几个方向：

> 1 独立的列。索引列不能是表达式的一部分，也不能是函数的参数。
>
> 2 前缀索引和索引选择性。这二者实际上是相互制约的关系，制约条件在于前缀的长度。一般应选择足够长的前缀以保证较高的选择性（保证查询效率），同时又不能太长以便节省空间。

3 尽量使用覆盖索引。覆盖索引是指一个索引包含所有需要查询的字段的值，这样在查询时只需要扫描索引而无须再去读取数据行，从而极大的提高性能。

4 使用索引扫描来做排序。要知道，扫描索引本身是很快的，在设计索引时，可尽可能的使用同一个索引既满足排序，又满足查找行数据。

最后，在建立索引时给几个小贴士：

1 优先使用自增key作为主键

2 记住最左前缀匹配原则

3 索引列不能参与计算

4 选择区分度高的列作索引

5 能扩展就不要新建索引

6 总结

索引的概念和原理是我们在了解和精通数据库过程中无法逃避的重点，而事实上建立高性能的索引对实际的应用场景也具有重要意义。本文的目的依旧是由浅入深的介绍索引这么个东西，从概念到实现再到最终的优化策略。

以上，希望大家都能有所收获，我们下篇见。

7 Reference

- 高性能 MySQL，Baron Schwartz 等人著，电子工业出版社
- 码海系列文章
- https://www.jianshu.com/p/9e9aca844c13
- https://www.runoob.com/mysql/mysql-index.html
- https://www.cnblogs.com/Aiapple/p/5693239.html
- https://blog.csdn.net/tongdanping/article/details/79878302
- https://www.cnblogs.com/igoodful/p/9361500.html

往期干货笔记整理

- **熬夜肝了个Linux速查备忘手册.pdf**
- **我的浏览器收藏夹大公开**
- **数据结构和算法刷题笔记.pdf下载**
- **LeetCode算法刷题C/C++版答案pdf下载**
- **LeetCode算法刷题Java版答案pdf下载**
- **找工作简历模板大分享.doc下载**
- **Java基础核心知识大总结.pdf 下载**
- **C/C++常见面试题（含答案）下载**
- **设计模式学习笔记.pdf下载**
- **Java后端开发学习路线+知识点总结**
- **前端开发学习路线+知识点总结**
- **大数据开发学习路线+知识点总结**
- **C/C++(后台)学习路线+知识点总结**
- **嵌入式开发学习路线+知识点总结**