索引

索引的意义

```
SELECT * FROM Employee

WHERE Employee_Name = 'Jesus'
```

如果表中没有索引会发生什么？

一旦我们运行这个查询, 数据库不得不**遍历Employee表中的每一行并确定雇员的名字（Employee_Name）是否为 'Jesus'**, 这就是所谓的**全表扫描**。

数据库索引是怎样提升性能的？

这就是索引派上用场的时候。使用索引的全部意义就是通过**缩小一张表中需要查询的记录/行的数目来加快搜索的速度。**

什么是索引？

一个索引是存储的表中一个特定列的值数据结构（最常见的是B+Tree）。**索引是在表的列上创建**。所以，要记住的关键点是索引包含一个表中列的值，并且这些值存储在一个数据结构中。**总之：索引是一种数据结构 。**

索引是怎么提升性能的？

因为索引基本上是用来存储列值的数据结构，这使查找这些列值更加快速。如果索引使用最常用的数据结构-B-Tree-那么其中的数据是**有序的**。有序的列值可以极大的提升性能。下面解释原因。

假设我们在 Employee_Name这一列上创建一个B-Tree索引。这意味着当我们用之前的SQL查找姓名是'Jesus'的雇员时，不需要再扫描全表。而是用索引查找去查找名字为'Jesus'的雇员，因为索引已经按照按字母顺序排序。索引**已经排序**意味着查询一个名字会快很多，因为名字首字母为'J'的员工都是排列在一起的(在使用聚簇索引的情况下)。另外重要的一点是，索引同时存储了表中相应行的指针以获取其他列的数据

数据库索引里究竟存的是什么？

我们已经知道数据库索引是创建在表的某列上的，并且存储了这一列的所有值。但是，需要理解的重点是**数据库索引并不存储这个表中其他列（字段）的值**(因为使用的是B+TREE结构)。举例来说，如果我们在Employee_Name列创建索引，那么列Employee_Age和Employee_Address上的值并不会存储在这个索引当中。如果我们确实把其他所有字段也存储在个这个索引中，那就成了拷贝一整张表做为索引-这样会占用太大的空间而且会十分低效。事实上,**索引存储了指向表中某一行的指针**

索引数据结构的选择

1. 哈希索引

存储引擎对所有的索隐列计算出一个哈希码，将哈希码存储在索引中，同时哈希表中保存每个数据行的指针。这样，对于此种索引查找速度是非常快的。出现哈希值碰撞的话，索引会以链表的形式存放多个记录指针到同一个哈希条目中。

name	age
Jane	28
Peter	20
David	30

假设使用假想的哈希函数f()，生成对应的设想值：
`f('Jane') = 2323`
`f('Peter') = 2456`
`f('David') = 2400`

则哈希索引的数据结构如下：

槽(slot)	值(value)
2323	指向第1行指针
2400	指向第3行指针
2456	指向第2行指针

对于`select * from user where`name`= 'Jane'`那么直接先算`Jane`的哈希值，然后根据`Jane`的hash值2323去找到对应的第一行数据，查询速度相对于B-Tree索引是要快，但是也有一些局限.

字段值所对应的数组下标是哈希算法随机算出来的，所以可能出现**哈希冲突**。

那么对于这样一个索引结构，现在来执行下面的sql语句：

> select * from sanguo where name='鸡蛋'

可以直接对'鸡蛋'按哈希算法算出来一个数组下标，然后可以直接从数据中取出数据并拿到所对应那一行数据的地址，进而查询那一行数据， 那么如果现在执行下面的sql语句：

> select * from sanguo where name > '鸡蛋'

则无能为力，因为哈希表的特点就是**可以快速的精确查询，但是不支持范围查询**。

如果做成了索引，那速度也是很慢的，要全部扫描。

Hash表在哪些场景比较适合？

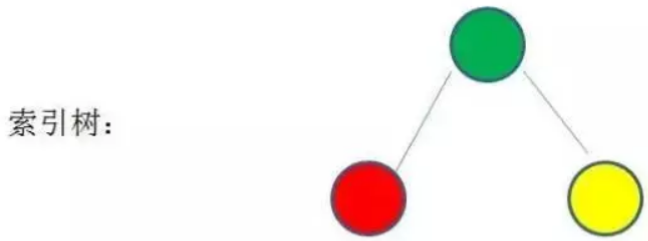
等值查询的场景，就只有KV（Key，Value）的情况，例如Redis、Memcached等这些中间件。

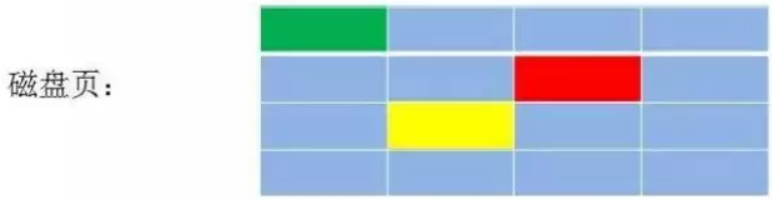
2. **为什么不用(完全平衡)二叉树（时间复杂度都是O(logN)）？**

   其实从算法逻辑上来讲，BST的查找速度和比较次数都是最小的。 但是，我们不得不考虑一个现实问题: **磁盘IO**

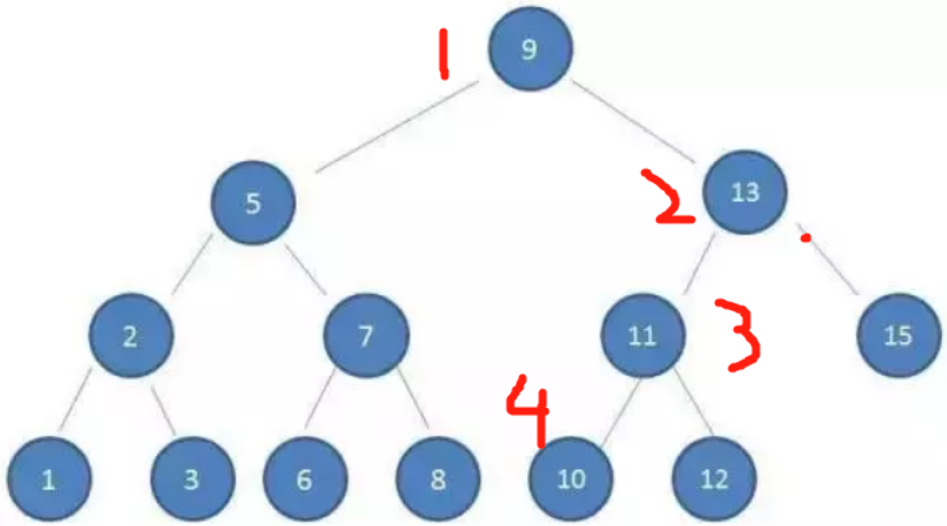
   数据库索引是存储在磁盘上的,当数据量比较大的时候,索引的大小甚至能达到几个G

   当我们利用索引查询的时候,不可能把索引加到内存,能做的只有逐个架子啊磁盘页,磁盘页对应着索引树的节点





如果我们利用BST作为索引结构,流程如下



我们可以很明显地看到,在最坏情况下,磁盘IO的次数等于索引树的高度.既然如此,为了减少磁盘IO的次数,就需要把原本"瘦高"的BST变得"矮胖",这就是B-TREE的特征之一

3. **B-TREE**

B树是一种**多路平衡查找树**，它的**每一个节点最多包含k个孩子.k 被称为B树的阶。k的大小取决于磁盘页的大小**

下面来具体介绍一下B-树（Balance Tree），一个m阶的B树具有如下几个特征：

1.根结点至少有两个子女。

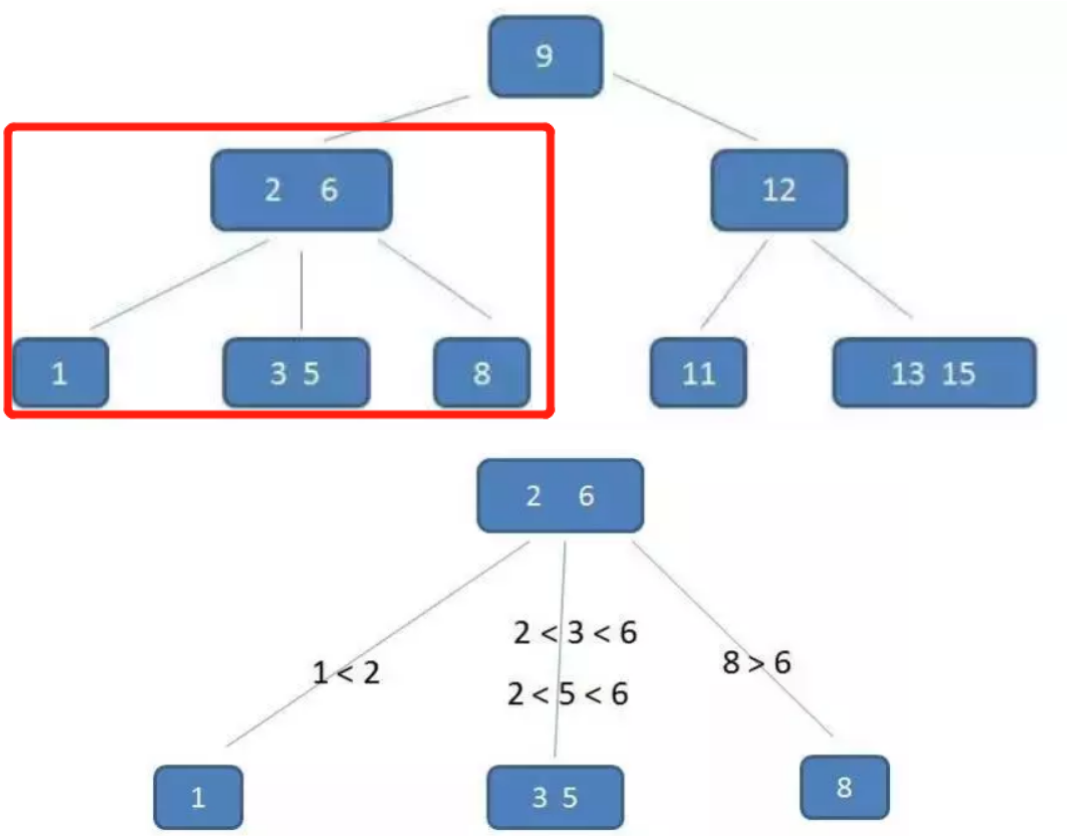
2.每个**非叶子节点**都包含k-1个元素**和k个孩子**，其中 m/2 <= k <= m

3.每一个**叶子节点**都包含k-1个元素，其中 m/2 <= k <= m

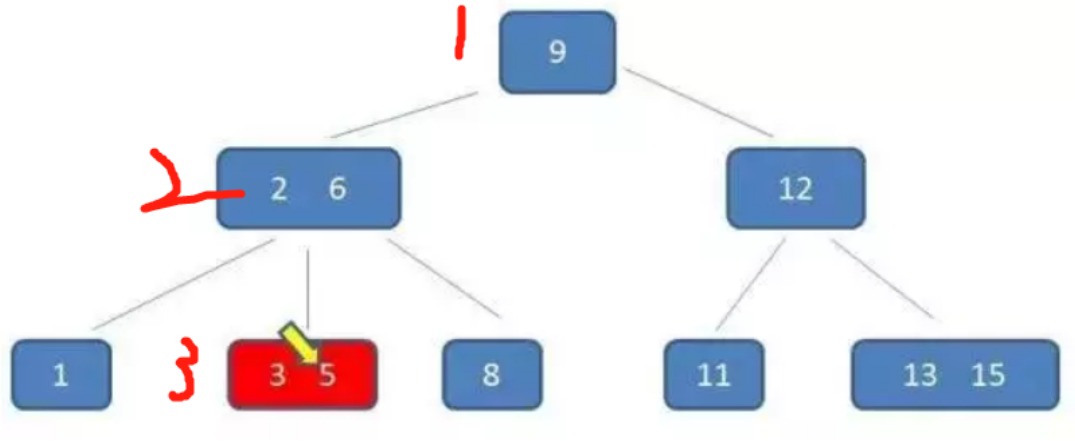
4.所有的叶子结点都位于同一层。

5.每个节点中的元素从小到大排列，节点当中k-1个元素正好是k个孩子包含的元素的值域分划。

以三阶B-TREE为例:



举例来说没如果我们要寻找5这个元素,最多只要3次磁盘IO



B-TREE主要应用于文件系统以及部分数据库索引，比如著名的非关系型数据库**MongoDB**。

而大部分关系型数据库，比如Mysql, 则使用**B+树**作为索引。

4. **B+TREE**

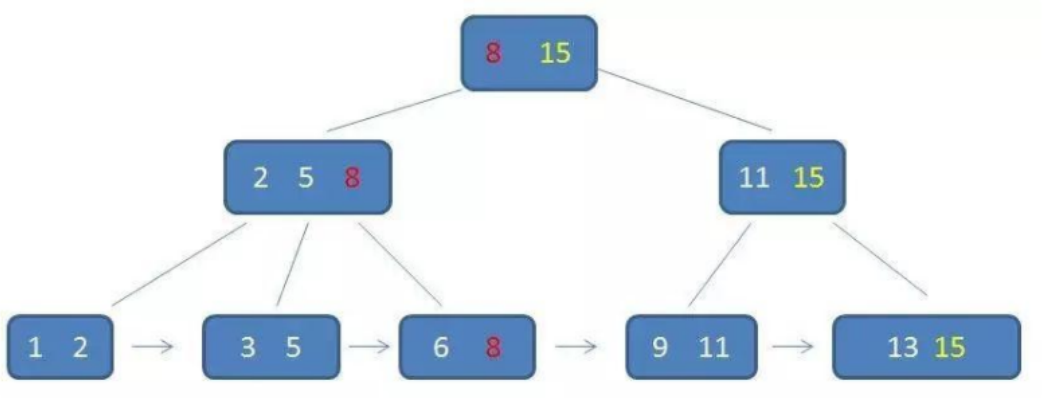
B+TREE是B-TREE的一种变体,有着更好的查询性能

一个m阶的B+树具有如下几个特征：

1.有k个子树的中间节点包含有k个元素（B树中是k-1个元素），每个元素**不保存数据，只用来索引，所有数据都保存在叶子节点。**

2.所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小**自小而大顺序链接。**

3.所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。



我们可以看到,B+TREE中节点间含有重复元素(父节点元素都出现在子节点中,是子节点的最大(或最小)元素),且叶子节点用指针连接在一起,并且每个叶子节点都带有指向下一个节点的指针,形成了有序链表.

B+树相对于B树的优点

1. 在B+TREE中,只有叶子节点含有索引指向的数据记录(卫星数据),**非叶子节点存储的是索引**,而B-TREE中每个节点都有.所以对于B+数来说,虽然查询过程可能一致,但是由于中间节点没有卫星数据,同样大小的磁盘页可以容纳更多元素,这就意味着，数据量相同的情况下, B+树的结构比B-树更加“矮胖”，因此查询时IO次数也更少。
2. 其次，B+树的查询**必须最终查找到叶子节点**，而B-树只要找到匹配元素 即可，无论匹配元素处于中间节点还是叶子节点,因此,B-TREE性能并不稳定,而B+树是稳定的
3. 范围查询时,B-树只能做烦琐的中序遍历.而B+树查找到叶子节点之后,只要通过链表指针遍历就可以找到

索引分类

1. 主键索引&辅助索引&唯一索引

在索引的分类中，我们可以按照索引的键是否为主键来分为“主索引”和“辅助索引”，**使用主键键值建立的索引称为“主索引”，其它的称为“辅助索引”。因此主索引只能有一个，辅助索引可以有很多个。**

唯一索引：用来建立索引的列的值必须是唯一的，允许空值

2. 聚簇索引和非聚簇索引

聚簇索引的解释是:**聚簇索引的顺序就是数据的物理存储顺序**

非聚簇索引的解释是:索引顺序与数据物理排列顺序无关.索引项顺序存储，但索引项对应的内容却是随机存储的；

理解聚集索引和非聚集索引可通过对比汉语字典的索引。汉语字典提供了两类检索汉字的方式，第一类是拼音检索（前提是知道该汉字读音），比如拼音为cheng的汉字排在拼音chang的汉字后面，根据拼音找到对应汉字的页码（因为按拼音排序，二分查找很快就能定位），这就是我们通常所说的字典序；第二类是部首笔画检索，根据笔画找到对应汉字，查到汉字对应的页码。**拼音检索就是聚集索引，因为存储的记录（数据库中是行数据、字典中是汉字的详情记录）是按照该索引排序的；笔画索引，虽然笔画相同的字在笔画索引中相邻，但是实际存储页码却不相邻。**

Innodb存储引擎中行记录就是按照聚集索引维度顺序存储的，Innodb的表也称为索引表；因为行记录只能按照一个维度进行排序，所以一张表只能有一个聚集索引(一般为主键,如果表中没有显示指定主键，则会选择表中的第一个不允许为NULL的唯一索引，如果还是没有的话，就采用Innodb存储引擎为每行数据内置的6字节ROWID作为聚集索引。)。

MyISAM和InnoDB索引实现的不同（存储结构）

MyISAM——非聚簇索引

- MyISAM存储引擎采用的是非聚簇索引，**非聚簇索引的主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，他们的叶子结点的key都存储指向键值对应的数据的物理**地址。**
- 非聚簇索引的数据表和索引表是分开存储的。
- 非聚簇索引中的数据是根据数据的插入顺序保存。因此非聚簇索引更适合单个数据的查询。插入顺序不受键值影响。

最开始我一直不懂既然非聚簇索引的主索引和辅助索引指向相同的内容，为什么还要辅助索引这个东西呢，后来才明白索引不就是用来查询的吗，用在那些地方呢，不就是WHERE和ORDER BY 语句后面吗，**那么如果查询的条件不是主键怎么办呢，这个时候就需要辅助索引了。**

InnoDB——聚簇索引

- 聚簇索引的主索引的叶子结点存储的是键值对应的**数据本身**，辅助索引(二级索引)的叶子结点存储的是键值对应的数据的主键键值。因此主键的值长度越小越好，类型越简单越好。
- 聚簇索引的数据和主键索引存储在一起。
- **聚簇索引的数据是根据主键的顺序保存。因此适合按主键索引的区间查找**，可以有更少的磁盘I/O，加快查询速度。但是也是因为这个原因，聚簇索引的插入顺序最好按照主键单调的顺序插入，否则会频繁的引起页分裂，严重影响性能。
- 在InnoDB中，查询时如果只需要查找索引的列，就尽量不要加入其它的列，这样会提高查询效率。

使用主索引的时候，更适合使用聚簇索引，因为聚簇索引只需要查找一次，而非聚簇索引在查到数据的地址后，还要进行一次I/O查找数据。

- 聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：**首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录(也就是我们经常说的回表)。**
- 因为聚簇辅助索引存储的是主键的键值，因此可以在数据行移动或者页分裂的时候降低成本，因为这时不用维护辅助索引。但是由于主索引存储的是数据本身，因此聚簇索引会占用更多的空间。
- 聚簇索引在插入新数据的时候比非聚簇索引慢很多，因为插入新数据时需要检测主键是否重复，这需要遍历主索引的所有叶节点，而非聚簇索引的叶节点保存的是数据地址，占用空间少，因此分布集中，查询的时候I/O更少，但聚簇索引的主索引中存储的是数据本身，数据占用空间大，分布范围更大，可能占用好多的扇区，因此需要更多次I/O才能遍历完毕。

图 5-9 是描述 InnoDB 和 MyISAM 如何存放表的抽象图。从图 5-9 中可以很容易看出 InnoDB 和 MyISAM 保存数据和索引的区别。

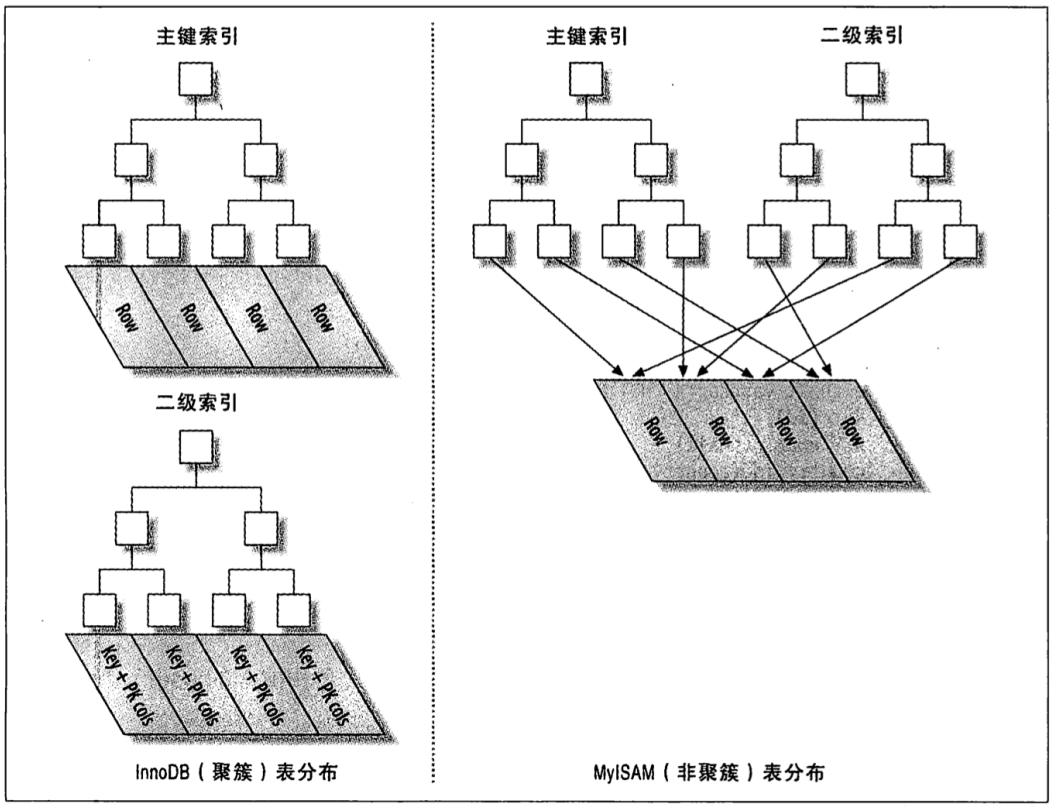


图5-9：聚簇和非聚簇表对比图

比较

- 事务：InnoDB 是事务型的，可以使用 Commit 和 Rollback 语句。
- 并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 外键：InnoDB 支持外键。
- 备份：InnoDB 支持在线热备份。
- 崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。
- 其它特性：MyISAM 支持压缩表和空间数据索引。

3. 全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引使用**倒排索引**实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

索引优化

1. 独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用 actor_id 列的索引：

```
SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

2. 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面的语句中，最好把 actor_id 和 film_id 设置为多列索引。

```
SELECT film_id, actor_ id FROM sakila.film_actor
WHERE actor_id = 1 AND film_id = 1;
```

3. 索引列的顺序

让选择性最强的索引列放在前面。

索引的选择性是指：不重复的索引值和记录总数的比值。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，每个记录的区分度越高，查询效率也越高。

例如下面显示的结果中 customer_id 的选择性比 staff_id 更高，因此最好把 customer_id 列放在多列索引的前面。

```
SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
COUNT(*)
FROM payment;
```

```
   staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
               COUNT(*): 16049
```

4. 前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。

前缀长度的选取需要根据索引选择性来确定。

5. 覆盖索引

索引包含所有需要查询的字段的值。

具有以下优点：

- 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
- 一些存储引擎（例如 MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
- 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引(避免回表问题)。

6.最左匹配原则：

- 索引可以简单如一个列 (a)，也可以复杂如多个列 (a,b,c,d)，即**联合索引。**
- 如果是联合索引，那么key也由多个列组成，同时，索引只能用于查找key是否**存在（相等）**，遇到范围查询 (>、<、between、like左匹配)等就**不能进一步匹配**了，后续退化为线性查找。
- 因此，**列的排列顺序决定了可命中索引的列数。**

例子：

- 如有索引 (a,b,c,d)，查询条件 a=1 and b=2 and c>3 and d=4，则会在每个节点依次命中a、b、c，无法命中d。(c已经是范围查询了，d肯定是排不了序了)

比如想要实现分页显示功能：
正常sql语句
SELECT * FROM table WHERE 查询条件 ORDER BY 排序条件 LIMIT ((页码-1)*页大小),页大小;

如果一共100页，1页20行记录，我想直接显示最后一页的数据，使用上面的语句实际上是要将所有数据从磁盘里读出来，然后排个序，然后取出最后20条数据。

优化结果

SELECT id FROM table WHERE 查询条件 ORDER BY 排序条件 LIMIT ((页码-1)*页大小),页大小;
SELECT * FROM table WHERE ID in （?) ORDER BY 排序条件 ;

拆分成这两句话执行

如果是查询条件有索引，
原来语句就是普通索引找到主键id，然后回表找记录
后面的语句，第一条是普通索引找id（与原来不同的是*和id，加载的数据量不一样），第二条使用主键索引，效率高

然后我们一般都是看explain sql的时候，是否命中了索引

如果没有命中，看是否有必要加上索引，加索引的条件还得看这个字段的区分度高不高，比如一个status字段的0/1两种状态数据量大概一半一半，其实就没什么必要加索引了