**索引常见面试题**

## 什么是索引？

索引就是**帮助存储引擎快速获取数据的一种数据结构**，形象的说就是**索引是数据的目录**。

**存储引擎就是如何存储数据、如何为存储的数据建立索引和如何更新、查询数据等技术的实现方法。**MySQL 存储引擎有 MyISAM 、InnoDB、Memory，其中 InnoDB 是在 MySQL 5.5 之后成为默认的存储引擎。

## 索引的分类

* 按「**数据结构**」分类：**B+tree索引、Hash索引、Full-text索引**。
* 按「**物理存储**」分类：**聚簇索引（主键索引）、二级索引（辅助索引）**。
* 按「**字段特性**」分类：**主键索引、唯一索引、普通索引、前缀索引**。
* 按「**字段个数**」分类：**单列索引、联合索引**。

### 按数据结构分类

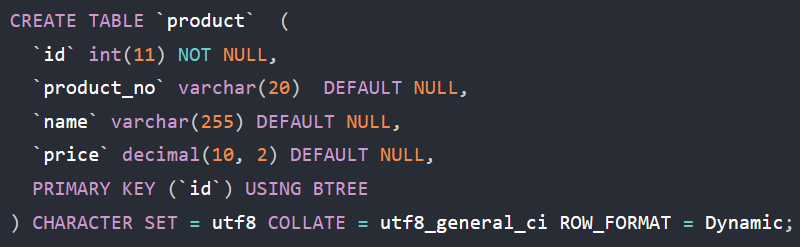
每一种存储引擎支持的索引类型各不相同：

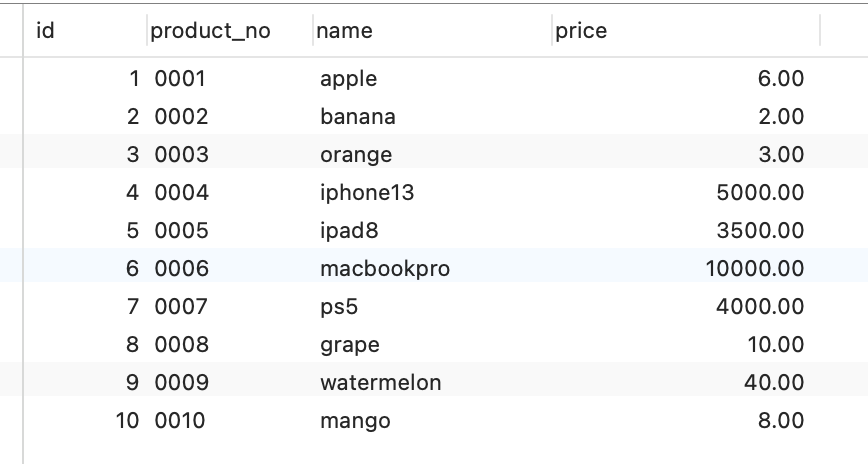


**B+Tree 索引类型是 MySQL 存储引擎采用最多的索引类型，创建的主键索引和二级索引默认使用的是 B+Tree 索引**。

B+Tree 是一种多叉树，**叶子节点才存放数据，非叶子节点只存放索引**，而且**每个节点里的数据是按主键顺序存放的**。每一层**父节点的索引值都会出现在下层子节点的索引值中**，因此在叶子节点中，包括了所有的索引值信息，并且每一个叶子节点都有两个指针，分别指向下一个叶子节点和上一个叶子节点，形成一个双向链表。

举例：

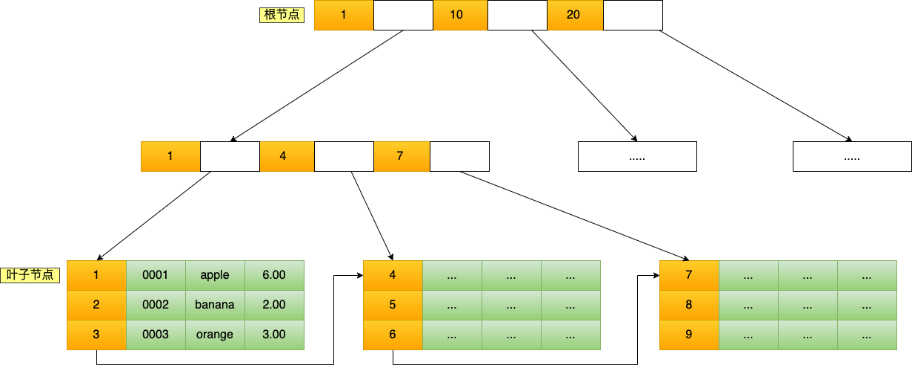




#### 通过主键查询商品数据的过程



主键索引的 B+Tree 如图所示（**叶子节点之间实际上是双向链表**）：



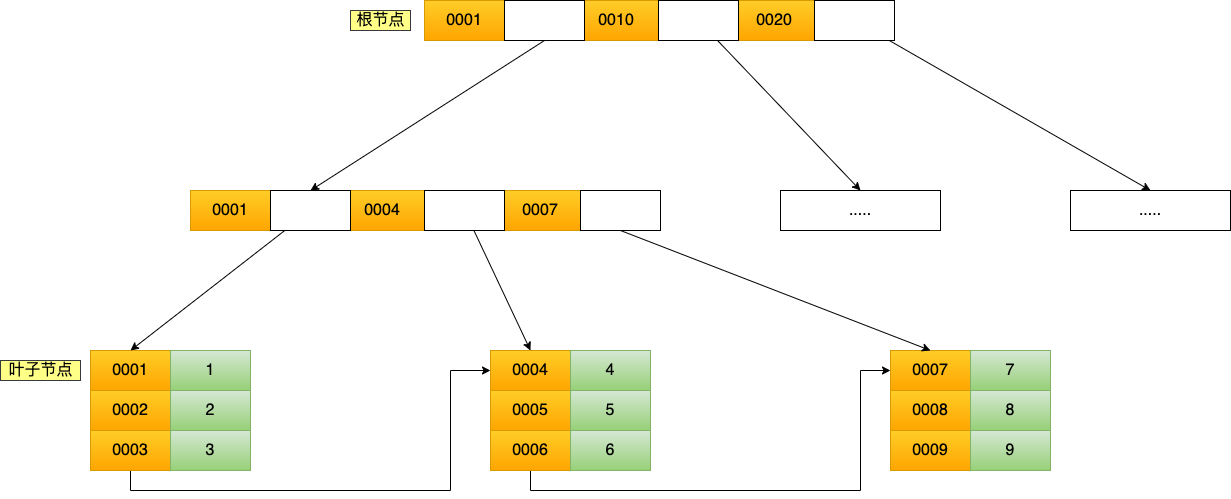
数据库的索引和数据都是存储在硬盘的，**读取一个节点可以看作一次磁盘 I/O 操作**。B+Tree 存储千万级的数据只需要 **3-4 层高度**就可以满足，这意味着从千万级的表查询目标数据最多需要 3-4 次磁盘 I/O，所以**B+Tree 相比于 B 树和二叉树来说，最大的优势在于查询效率很高，因为即使在数据量很大的情况，查询一个数据的磁盘 I/O 依然维持在 3-4次。**

#### 通过二级索引查询商品数据的过程



主键索引的 B+Tree 和二级索引的 B+Tree 区别：

* **主键索引**的 B+Tree 的叶子节点存放的是**实际数据**，**所有完整的用户记录**都存放在主键索引的 B+Tree 的叶子节点里；
* **二级索引**的 B+Tree 的叶子节点存放的是**主键值**，而不是实际数据。



先检索二级索引中的 B+Tree 的索引值（商品编码，product\_no），**找到对应的叶子节点**，**然后获取主键值**，然后再通过主键索引中的 B+Tree 树查询到对应的叶子节点，然后获取整行数据。**这个过程叫「回表」，也就是说要查两个 B+Tree 才能查到数据**。

当查询的数据是能在二级索引的 B+Tree 的叶子节点里查询到，这时就不用再查主键索引查。**这种在二级索引的 B+Tree 就能查询到结果的过程就叫作「覆盖索引」，也就是只需要查一个 B+Tree 就能找到数据**。



#### 为什么 MySQL InnoDB 选择 B+tree 作为索引的数据结构？

***1、B+Tree vs B Tree***

**B+Tree 只在叶子节点存储数据，而 B 树 的非叶子节点也要存储数据**，所以 B+Tree 的**单个节点的数据量更小**，**在相同的磁盘 I/O 次数下，就能查询更多的节点**。另外，B+Tree 叶子节点采用的是**双链表**连接，适合 MySQL 中常见的**基于范围的顺序查找**，而 B 树无法做到这一点。

***2、B+Tree vs 二叉树***

对于有 N 个叶子节点的 B+Tree，其搜索复杂度为O(logdN)，其中 d 表示节点允许的最大子节点个数为 d 个。

在实际的应用当中， **d 值是大于100的**，这样就保证了，即使数据达到千万级别时，B+Tree 的高度依然维持在 **3~4 层**左右，也就是说一次数据查询操作只需要做 3~4 次的磁盘 I/O 操作就能查询到目标数据。

而二叉树的每个父节点的儿子节点个数只能是 2 个，意味着其搜索复杂度为 O(log2N)，这已经比 B+Tree 高出不少，因此二叉树检索到目标数据所经历的**磁盘 I/O 次数**要更多。

***3、B+Tree vs Hash***

Hash 在做**等值查询**的时候效率很快，搜索复杂度为 O(1)。但是 **Hash 表不适合做范围查询**，它更适合做等值的查询，这也是 B+Tree 索引要比 Hash 表索引有着更广泛的适用场景的原因。

### 按物理存储分类

从物理存储的角度来看，索引分为**聚簇索引（主键索引）**、**二级索引（辅助索引）：**

* 主键索引的 B+Tree 的叶子节点存放的是**实际数据**，**所有完整的用户记录**都存放在主键索引的 B+Tree 的叶子节点里；
* 二级索引的 B+Tree 的叶子节点存放的是**主键值**，而不是实际数据。

在创建表时，InnoDB 存储引擎会根据不同的场景选择不同的列作为聚簇索引：

* 如果有**主键**，默认会使用主键作为聚簇索引的索引键（key）；
* 如果没有主键，就选择**第一个不包含 NULL 值的唯一列**作为聚簇索引的索引键（key）；
* 在上面两个都没有的情况下，InnoDB 将**自动生成一个隐式自增 id 列**作为聚簇索引的索引键（key）；

如果**查询的数据能在二级索引里查询的到**，那么就不需要回表，这个过程就是**覆盖索引**。如果**查询的数据不在二级索引里**，就会先检索二级索引，找到对应的叶子节点，获取到主键值后，然后再检索主键索引，就能查询到数据了，这个过程就是**回表**。

### 按字段特性分类

从字段特性的角度来看，索引分为**主键索引**、**唯一索引**、**普通索引**、**前缀索引**。

#### 主键索引

建立在**主键字段**上的索引，通常在创建表的时候一起创建，**一张表最多只有一个主键索引**，索引列的值不允许有空值。

#### 唯一索引

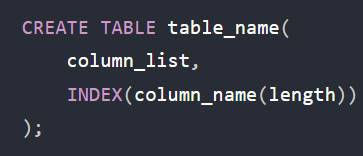
建立在 **UNIQUE 字段**上的索引，**一张表可以有多个唯一索引**，索引列的值必须唯一，但是允许有空值。

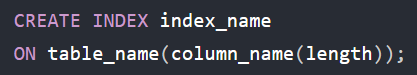
#### 普通索引

普通索引就是建立在**普通字段**上的索引，既不要求字段为主键，也不要求字段为 UNIQUE。

#### 前缀索引

前缀索引是指对**字符类型字段的前几个字符**建立的索引，而不是在整个字段上建立的索引，前缀索引可以建立在字段类型为 **char、 varchar、binary、varbinary** 的列上。

创建表时：

建表后：

### 按字段个数分类

从字段个数的角度来看，索引分为**单列索引**、**联合索引（复合索引）**

* 建立在**单列上的索引**称为单列索引，比如主键索引；
* 建立在**多列上的索引**称为联合索引；

使用联合索引时，存在**最左匹配原则**，也就是**按照最左优先的方式进行索引的匹配**，如果不遵循「最左匹配原则」，联合索引会失效

举例：

创建了一个 (a, b, c) 联合索引

① 联合索引有效：

* where a=1；
* where a=1 and b=2 and c=3；
* where a=1 and b=2；

**注意a，b，c的顺序无所谓（查询优化器会优化），重要的是有b必须有a，有c必须有a和b**

② 联合索引失效：

* where b=2；
* where c=3；
* where b=2 and c=3；

原因：**b 和 c 是全局无序，局部相对有序的**

#### 联合索引范围查询

注意：**并不是查询过程使用了联合索引查询，就代表联合索引中的所有字段都用到了联合索引进行索引查询**，也就是可能存在**部分字段用到联合索引、部分字段没有用到联合索引**的 B+Tree 的情况。

Q1: select \* from t\_table where a > 1 and b = 2，联合索引（a, b）哪一个字段用到了联合索引的 B+Tree？

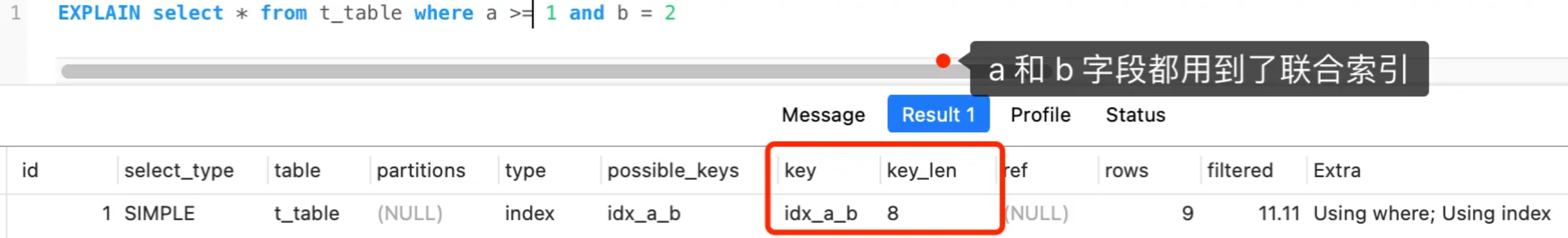
**只有 a 字段用到了联合索引进行索引查询，而 b 字段并没有使用到联合索引。因为符合 a > 1 条件的二级索引记录的范围里，b 字段的值是无序的**。



Q2: select \* select t\_table where a >= 1 and b = 2，联合索引（a, b）哪一个字段用到了联合索引的 B+Tree？

**a 和 b 字段都用到了联合索引进行索引查询**

虽然在符合 a>= 1 条件的二级索引记录的范围里，b 字段的值是「无序」的，**但是对于符合 a = 1 的二级索引记录的范围里，b 字段的值是「有序」的。**



Q3: select \* select t\_table where a between 2 and 8 and b = 2，联合索引（a, b）哪一个字段用到了联合索引的 B+Tree？

**MySQL 的 BETWEEN 包含 value1 和 value2 边界值**，所以类似于 Q2 查询语句，因此 **Q3 这条查询语句 a 和 b 字段都用到了联合索引进行索引查询**。

Q4: select \* from t\_user where name like 'j%' and age = 22，联合索引（name, age）哪一个字段用到了联合索引的 B+Tree？

虽然在符合前缀为 ‘j’ 的 name 字段的二级索引记录的范围里，age 字段的值是「无序」的，**但是对于符合 name = j 的二级索引记录的范围里，age字段的值是「有序」的。name和 age 字段都用到了联合索引进行索引查询**。

可以理解为：**字符串大于等于j（即字典序大于等于j）**

个人理解：如果使用’\_’而不是’%’，仍然可以都用到联合索引，相当于字符串大于等于’j[字典中的第一个字符]’

总结：**联合索引的最左匹配原则，在遇到范围查询（如 >、<）的时候，就会停止匹配，也就是范围查询的字段可以用到联合索引，但是在范围查询字段的后面的字段无法用到联合索引。注意，对于 >=、<=、BETWEEN（MySQL中）、like 前缀匹配的范围查询，并不会停止匹配**

#### 索引下推

执行 select \* from table where a > 1 and b = 2 语句时：

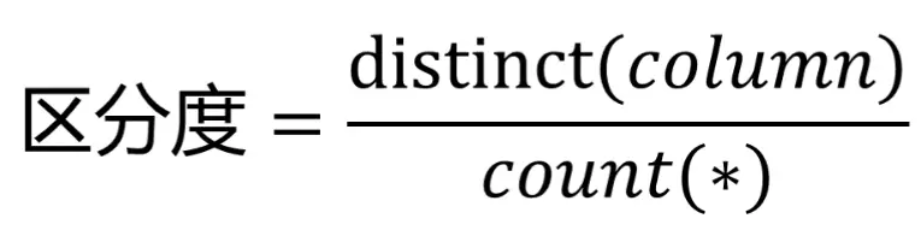
* 在 MySQL 5.6 之前，只能从 a字段大于1的第一个字段开始一个个回表，到「主键索引」上找出数据行，再对比 b 字段值。
* 而 MySQL 5.6 引入的**索引下推优化**（index condition pushdown)， **可以在联合索引遍历过程中，对联合索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数**。

执行计划里，出现了 **Extra 为 Using index condition**，那么说明使用了索引下推的优化

#### 索引区分度

建立联合索引时的**字段顺序**，对索引效率也有很大影响。**越靠前的字段被用于索引过滤的概率越高**，实际开发工作中**建立联合索引时，要把区分度大的字段排在前面，这样区分度大的字段越有可能被更多的 SQL 使用到**。

区分度就是某个字段 column 不同值的个数「除以」表的总行数，计算公式如下：



MySQL 还有一个查询优化器，查询优化器**发现某个值出现在表的数据行中的百分比**（惯用的百分比界线是"30%"）**很高**的时候，它一般会**忽略索引，进行全表扫描**。

#### 联合索引进行排序



① 如果**只用到 status 的索引，在得到结果后还要根据 create\_time 排序**，这时就要用文件排序 filesort，也就是在 SQL 执行计划中，Extra 列会出现 Using filesort。

② 利用索引的有序性，**在 status 和 create\_time 列建立联合索引，这样根据 status 筛选后的数据就直接是按照 create\_time 排好序的**，避免在文件排序，提高了查询效率。

## 什么时候需要 / 不需要创建索引？

优点：**提高查询速度**

缺点：

* 需要**占用物理空间**，数量越大，占用空间越大；
* 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增大；
* 会**降低表的增删改的效率**，因为每次增删改索引，B+ 树为了维护索引有序性，都需要进行动态维护。

#### 什么时候适用索引？

* 字段有**唯一性限制**的，比如商品编码；
* **经常用于 WHERE 查询条件的字段**，这样能够提高整个表的查询速度，如果查询条件不是一个字段，可以建立联合索引。
* **经常用于 GROUP BY 和 ORDER BY 的字段**，这样在查询的时候就不需要再去做一次排序了。

#### 什么时候不需要创建索引？

* **WHERE ，GROUP BY，ORDER BY 里用不到的字段**，索引的价值是快速定位，如果起不到定位的字段通常是不需要创建索引的，因为索引是会占用物理空间的。
* 字段中**存在大量重复数据**，不需要创建索引，比如性别字段。
* **表数据太少**的时候，不需要创建索引；
* **经常更新的字段**不用创建索引，比如电商项目的用户余额。索引字段频繁修改，由于要维护 B+Tree的有序性，那么就需要频繁的重建索引，这个过程是会影响数据库性能的。

## 有什么优化索引的方法？

### 前缀索引优化

使用某个字段中字符串的前几个字符建立索引

局限性：

① order by无法使用

② 无法用作覆盖索引

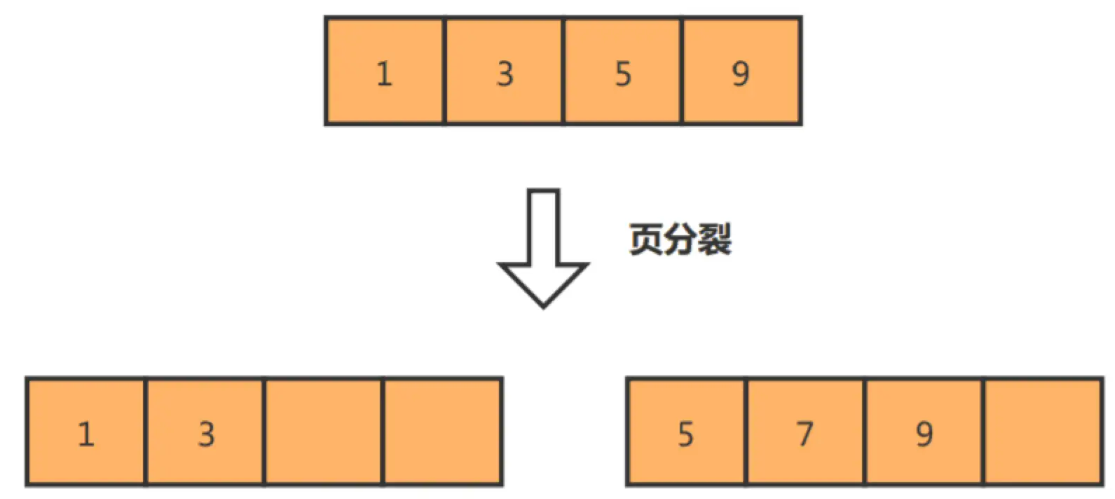
### 覆盖索引优化

SQL 中 query 的所有字段，在索引 B+Tree 的叶子节点上都能找得到的那些索引，从二级索引中查询得到记录，而不需要通过聚簇索引查询获得，可以**避免回表**的操作。

### 主键索引最好是自增的

**自增主键**：每次插入的新数据就会**按顺序添加**到当前索引节点的位置，**不需要移动已有的数据，当页面写满，就会自动开辟一个新页面**。

**非自增主键**：每次插入主键的索引值都是随机的，插入新的数据时可能会**插入到现有数据页中间的某个位置**，这将不得不**移动其它数据**来满足新数据的插入，**甚至需要从一个页面复制数据到另外一个页面**，我们通常将这种情况称为**页分裂**。**页分裂还有可能会造成大量的内存碎片，导致索引结构不紧凑，从而影响查询效率**。



另外，**主键字段的长度不要太大**，因为**主键字段长度越小，意味着二级索引的叶子节点越小（二级索引的叶子节点存放的数据是主键值），这样二级索引占用的空间也就越小**。

### 索引最好设置为 NOT NULL

索引列要设置为 NOT NULL 约束：

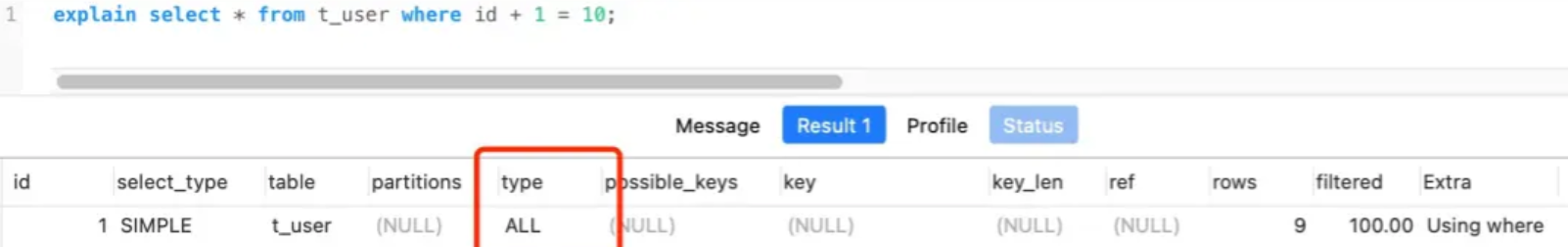
* 索引列存在 NULL 就会**导致优化器在做索引选择的时候更加复杂和难以优化**，因为可为 NULL 的列会使索引、索引统计和值比较都更复杂，比如进行索引统计时，**count 会省略值为NULL 的行**。
* NULL 值会占用物理空间（**NULL 值列表**）

### 防止索引失效

索引失效的情况：

* 当我们使用**左或者左右模糊匹配**的时候，也就是 like %xx 或者 like %xx%这两种方式都会造成索引失效；
* 当我们在查询条件中**对索引列做了计算、函数、类型转换操作**，这些情况下都会造成索引失效；
* **联合索引要能正确使用需要遵循最左匹配原则**，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。
* 在 WHERE 子句中，如果在 **OR前后包含非索引列**，那么索引会失效。

实际过程中，可能会出现其他的索引失效场景，这时我们就需要**查看执行计划**判断查询语句是否使用了索引。



* possible\_keys 字段表示**可能用到的索引**；
* key 字段表示**实际用的索引**，如果这一项为 NULL，说明没有使用索引；
* key\_len 表示**索引的长度**；
* rows 表示**扫描的数据行数**。
* type 表示**数据扫描类型**，需要重点看这个。
* extra 表示语句中**包含一些特定的操作**

常见扫描类型type的**执行效率从低到高的顺序为**：

* All（全表扫描）
* index（全索引扫描）：经常在**排序**时候出现
* range（索引范围扫描）： where 子句中使用 **< 、>、in、between** 等关键词
* ref（非唯一索引扫描）：采用了**非唯一索引，或者是唯一索引的非唯一性前缀**
* eq\_ref（唯一索引扫描）：使用**主键或唯一索引**，通常使用在**多表联查**中
* const（结果只有一条的主键或唯一索引扫描）：使用了主键或者唯一索引与**常量值**进行比较

**需要尽量让 SQL 查询可以使用到 range 这一级别及以上的 type 访问方式**。

常见extra：

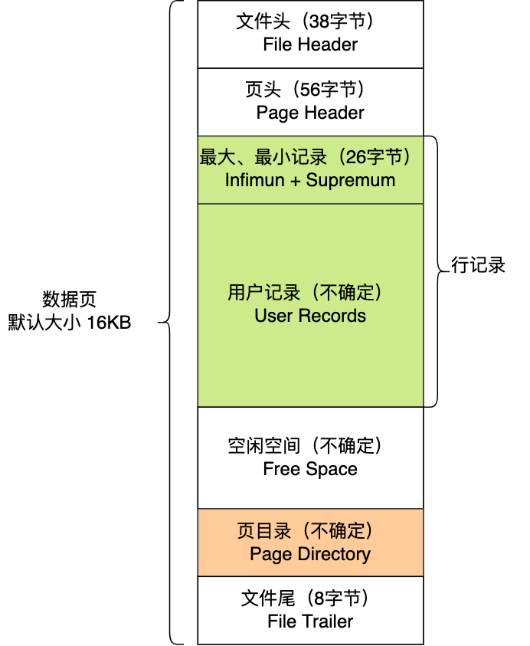
* Using filesort ：当查询语句中包含 **group by** 操作，而且**无法利用索引**完成排序操作的时候， 这时不得不选择相应的排序算法进行，甚至可能会通过文件排序，效率是很低的，所以要避免这种问题的出现。
* Using temporary：使了用**临时表**保存中间结果，MySQL 在对查询结果排序时使用临时表，**常见于排序 order by 和分组查询 group by**。效率低，要避免这种问题的出现。
* Using index：使用了**覆盖索引**，效率不错。

# 从数据页的角度看 B+ 树

## InnoDB 是如何存储数据的？

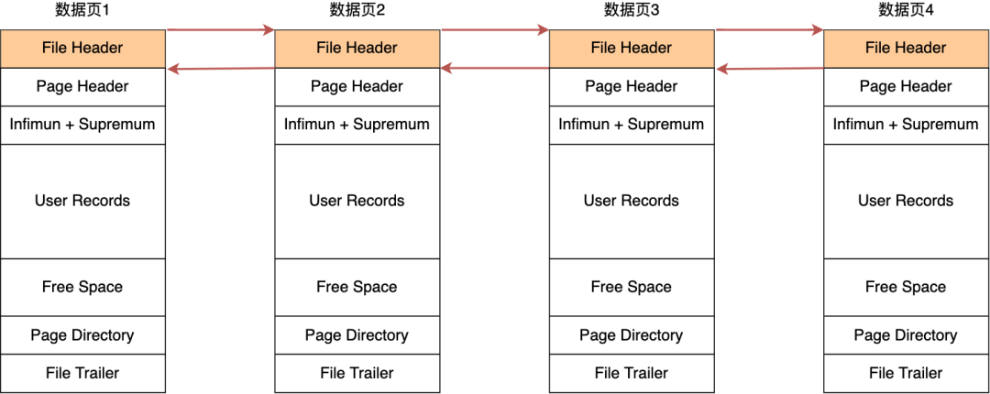
**InnoDB 的数据是按「数据页」为单位来读写的， 数据页的默认大小是 16KB**

数据页包括七个部分，结构如下图：

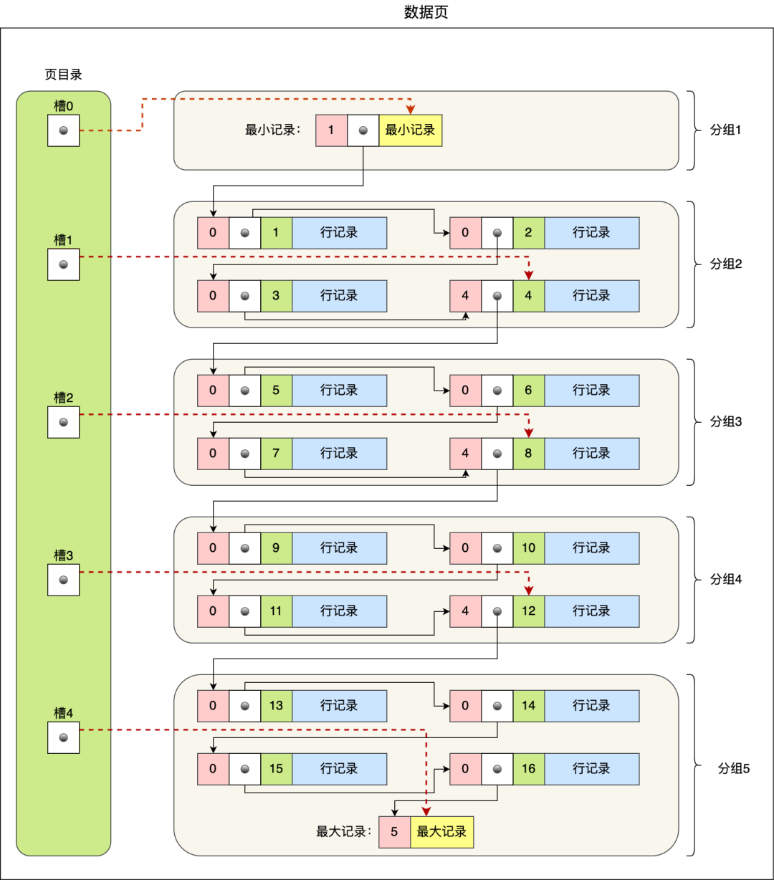




**数据页之间形成一个双向链表：**



**数据页中的记录按照「主键」顺序组成单向链表：**



特点就是**插入、删除非常方便，但是检索效率不高**，最差的情况下需要遍历链表上的所有节点才能完成检索。

页目录创建的过程如下：

1. 将所有的记录划分成几个组，这些记录包括最小记录和最大记录，但不包括标记为“已删除”的记录；
2. **每个记录组的最后一条记录就是组内最大的那条记录**，并且最后一条记录的头信息中会**存储该组一共有多少条记录**，作为 n\_owned 字段（上图中粉红色字段）
3. **页目录用来存储每组最后一条记录的地址偏移量**，这些地址偏移量会按照先后顺序存储起来，每组的地址偏移量也被称之为槽（slot），**每个槽相当于指针指向了不同组的最后一个记录**。

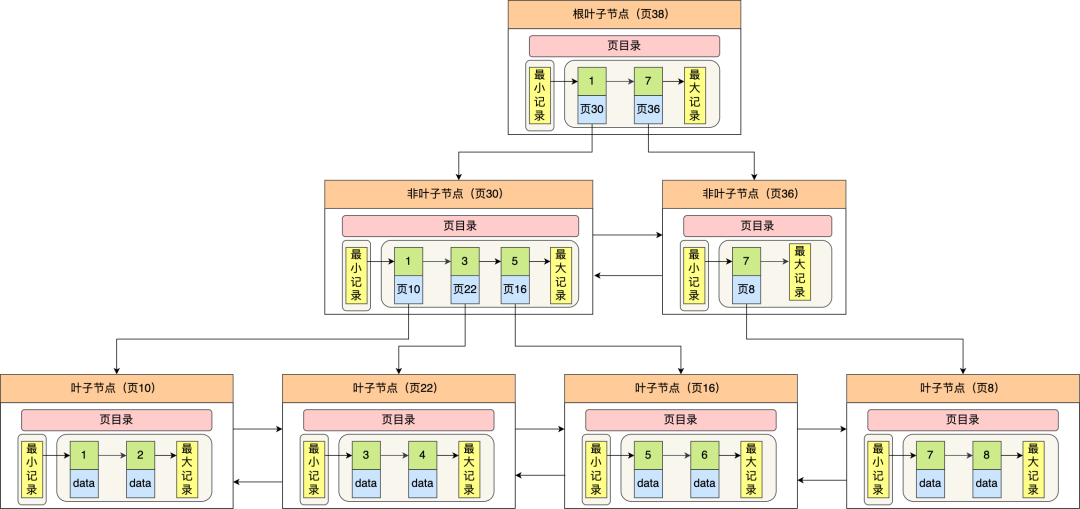
**页目录就是由多个槽组成的，槽相当于分组记录的索引**。然后，因为记录是按照「主键值」从小到大排序的，所以**我们通过槽查找记录时，可以使用二分法快速定位要查询的记录在哪个槽（哪个记录分组），定位到槽后，再遍历槽内的所有记录（先通过上一个分组的最后一个记录位置得到这一个分组的第一个位置，再遍历），找到对应的记录**，无需从最小记录开始遍历整个页中的记录链表。

槽内的记录就只有几条：

* 第一个分组中的记录只能有 1 条记录（只包含最小记录）；
* 最后一个分组中的记录条数范围只能在 1-8 条之间（包含最大记录）；
* 剩下的分组中记录条数范围只能在 4-8 条之间。

## B+ 树是如何进行查询的？

InnoDB 里的 B+ 树中的**每个节点都是一个数据页：**



根节点和非叶子节点中通过**二分法快速定位**到符合页内范围**包含查询值的页**，定位到该页后又会在该页内进行**二分法快速定位记录所在的分组（槽号）**，最后在**分组内进行遍历查找**。

# 为什么 MySQL 采用 B+ 树作为索引？

## 怎样的索引的数据结构是好的？

要设计一个适合 MySQL 索引的数据结构，至少满足以下要求：

* 能在**尽可能少的磁盘的 I/O 操作**中完成查询工作；
* 要能高效地**查询某一个记录**，也要能高效地执行**范围查找**；

## 二分查找树

**一个节点的左子树的所有节点都小于这个节点，右子树的所有节点都大于这个节点**

缺点：

① 极端情况：**当插入元素有序时，二叉查找树会退化成了一条链表，查找数据的时间复杂度变成了 O(n)**

② 没有B+树的双向链表结构，范围查询困难

## 自平衡二叉树（AVL树）

在二叉查找树的基础上，**每个节点的左子树和右子树的高度差不能超过 1**，使得查询操作的**时间复杂度就会一直维持在 O(logn)**

缺点：由于仍然是二叉树，高度还是很高，**I/O 操作次数多**

## B 树

B 树的每一个节点最多可以包括 M 个子节点和M-1个数据，M 称为 B 树的阶，所以 B 树就是一个**多叉树**。

缺点：

① **每个节点都包含数据（索引+记录）**，而用户的记录数据的大小很有可能远远超过了索引数据，这就**需要花费更多的磁盘 I/O 操作次数**来读到「有用的索引数据」。

② 查询过程中很多**不是查询目标的节点也被加载到内存中，占用内存资源**

③ 没有B+树的双向链表结构，范围查询困难

## B+ 树

* 叶子节点（最底部的节点）才会存放实际数据（索引+记录），非叶子节点只会存放索引；
* 所有索引都会在叶子节点出现，叶子节点之间构成一个有序链表；
* 非叶子节点的索引同时存在在子节点中，并且是在子节点中所有索引的最大（或最小）。
* 非叶子节点中有多少个子节点，就有多少个索引；

与B树的性能比较：

### 1、单点查询

B树：最快O(1)，最慢O(logn)，平均时间代价比 B+ 树稍快一些，但是**查询波动会比较大**

B+树：O(logn)。**数据量相同的情况下非叶子节点可以存放更多的索引，查询底层节点的磁盘 I/O次数会更少**。

### 2、插入和删除效率

B 树：没有冗余节点，删除节点的时候非常复杂，比如删除根节点中的数据，可能涉及复杂的树的变形

B+树：有大量的冗余节点，删除一个节点的时候可以直接从叶子节点中删除，甚至可以不动非叶子节点。**B+ 树的插入和删除效率更高**。

### 3、范围查询

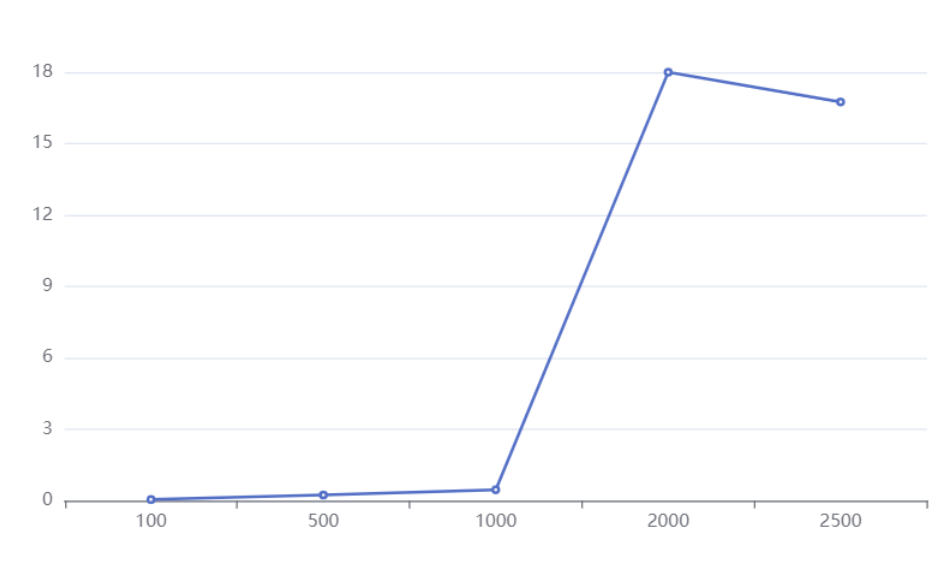
**B+ 树所有叶子节点间还有一个链表进行连接，这种设计对范围查找非常有帮助。**B 树没有。

存在**大量范围检索**的场景，适合使用 **B+树**，比如数据库。而对于**大量的单个索引查询**的场景，可以考虑 **B 树**，比如 nosql 的MongoDB。

# MySQL 单表不要超过 2000W 行，靠谱吗？

在保持相同的层级（相似查询性能）的情况下，在**行数据大小不同的情况下**，其实这个最大建议值也是不同的，而且影响查询性能的还有很多其他因素，比如，数据库版本，服务器配置，sql 的编写等等。

行的数据占用空间约为 1kB时，最大行数建议值约为2000w



当单表数据库到达某个量级的上限时，导致**内存无法存储其索引**，使得之后的 SQL 查询会产生**磁盘 IO**

# 索引失效有哪些？

## 对索引使用左或者左右模糊匹配

**like %xx** 或者 **like %xx%** 这两种方式都会造成索引失效

例如：如果使用 name like '%林' 方式来查询，因为查询的结果可能是「陈林、张林、周林」等之类的，所以不知道从哪个索引值开始比较，于是就只能通过全表扫描的方式来查询。

## 对索引使用函数

索引保存的是索引字段的**原始值**，而不是经过函数计算后的值，自然就没办法走索引了



从 MySQL 8.0 开始，索引特性增加了函数索引，即可以针对函数计算后的值建立一个索引：



## 对索引进行表达式计算

在查询条件中对索引进行表达式计算，也是无法走索引的



## 对索引隐式类型转换

隐式类型转换相当于对索引使用了一个CAST函数。

MySQL 的数据类型转换规则：**遇到字符串和数字比较的时候，会自动把字符串转为数字，然后再进行比较**。若字符串是以数字开头，并且全部都是数字，则转换的数字结果是整个字符串；部分是数字，则转换的数字结果是**截止到第一个不是数字的字符为止**。若字符串不是以数字开头，则转换的数字结果是 0。

官方的隐式转换说明：

1. 两个参数至少有一个是 NULL 时，比较的结果也是 NULL，例外是使用 <=> 对两个 NULL 做比较时会返回 1，这两种情况都不需要做类型转换

2. 两个参数都是字符串，会按照字符串来比较，不做类型转换

3. 两个参数都是整数，按照整数来比较，不做类型转换

4. 十六进制的值和非数字做比较时，会被当做二进制串

5. 有一个参数是 TIMESTAMP 或 DATETIME，并且另外一个参数是常量，**常量会被转换为 timestamp**

6. 有一个参数是 decimal 类型，如果另外一个参数是 decimal 或者整数，会将**整数转换为 decimal 后进行比较**，如果另外一个参数是浮点数，则会把 **decimal 转换为浮点数**进行比较

7. 所有其他情况下，两个参数都会被转换为浮点数再进行比较

## 联合索引非最左匹配

如果创建了一个 (a, b, c) 联合索引，如果查询条件是以下这几种，就可以匹配上联合索引：

* where a=1；
* where a=1 and b=2 and c=3；
* where a=1 and b=2；

需要注意的是，因为有查询优化器，所以 a 字段在 where 子句的顺序并不重要。

但是，如果查询条件是以下这几种，因为**不符合最左匹配原则**，所以就无法匹配上联合索引，联合索引就会失效:

* where b=2；
* where c=3；
* where b=2 and c=3；

**特别地**：where a = 1 and c = 3

MySQL 5.6 之前，只有a字段能用到索引，回表后在server层判断c字段

MySQL 5.6 及之后，使用**索引下推**，可以在存储引擎层进行索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，再返还给 Server 层

## WHERE 子句中的 OR

在 WHERE 子句中，OR前后存在非索引列，索引会失效。



# MySQL 使用 like “%x“，索引一定会失效吗？

如果一张表的**没有「非索引」字段**，就算使用like ‘%x’，也只要查询二级索引B+树就可以得到结果（由于是覆盖索引），没有发生索引失效

但是此时执行计划中的type是**index**（全索引扫描），如果是like ‘x %’ 则是**range**

**注意：like ‘xxx’ （没有使用模糊匹配）也是range，并不是ref，和 =’xxx’并不完全一样**

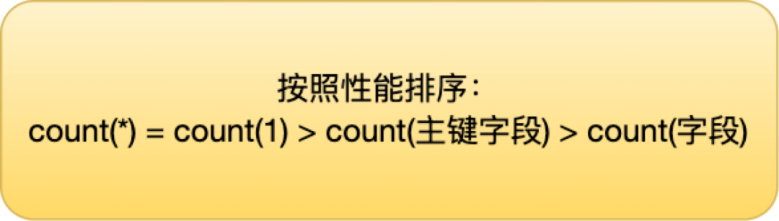
为什么选择全扫描二级索引树，而不扫描聚簇索引树呢？

**二级索引树**的记录东西很少，就**只有「索引列+主键值」**，而**聚簇索引**记录的东西会更多，比如聚簇索引中的叶子节点则记录了**主键值、事务 id、用于事务和 MVCC 的回滚指针以及所有的剩余列**。并且**不需要回表**

类似的：联合索引要遵循**最左匹配**才能走索引，但是如果数据库表中的字段都是索引的话，即使查询过程中，没有遵循最左匹配原则，也是走全扫描二级索引树(type=index)

# count(\*) 和 count(1) 有什么区别？哪个性能最好？

## 哪种 count 性能最好？



### count() 是什么？

count() 是一个聚合函数，函数的参数不仅可以是**字段名**，也可以是**其他任意表达式**，该函数作用是**统计符合查询条件的记录中，函数指定的参数不为 NULL 的记录有多少个**。通过 count 函数统计有多少个记录时，server 层会维护一个名叫 count 的变量。server 层会**循环向 InnoDB 读取一条记录**，如果 count 函数指定的参数不为 NULL，那么就会将变量 count 加 1，直到符合查询的全部记录被读完就退出循环。



统计t\_order 表中，name 字段不为 NULL 的记录有多少个



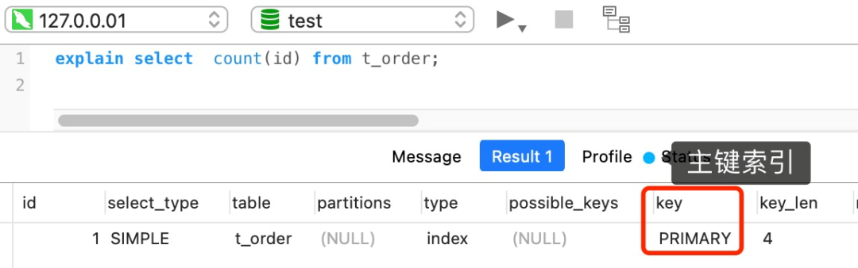
统计t\_order 表中，1 这个表达式不为 NULL 的记录有多少个。显然1永远都不是 NULL，所以这条语句其实是在统计 t\_order 表中有多少个记录。

### count(主键字段) 执行过程

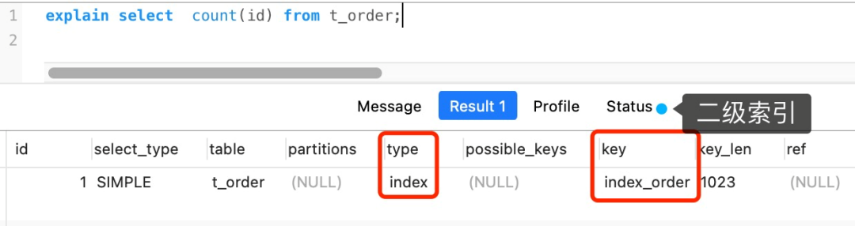
举例：

 （id 为主键）

① 若表中只有主键索引，没有二级索引，innoDB**遍历聚簇索引**



② 若有二级索引，InnoDB 循环**遍历二级索引**。



「优化器」**优先选择的是二级索引**，因为相同数量的二级索引记录可以比聚簇索引记录占用更少的存储空间。

### count(1) 执行过程

与count(主键字段)基本类似，**优先选择的是二级索引**

不同点：InnoDB 循环遍历聚簇索引（主键索引），将读取到的记录返回给 server 层，**但是不会读取记录中的任何字段的值**，因此**count(1) 执行效率会比 count(主键字段) 高一点**

### count(\*) 执行过程

这里的\*并不代表所有字段！

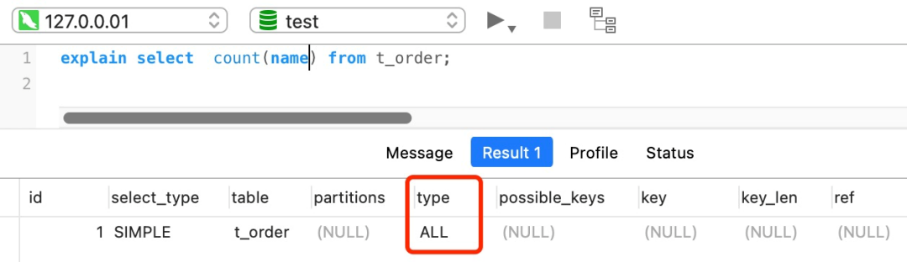
官方手册：

*InnoDB handles SELECT COUNT(\\*) and SELECT COUNT(1) operations* ***in the same way.******There is no performance difference.***

此外，MySQL 会对 count(\*) 和 count(1) 有个优化，如果有多个二级索引的时候，优化器会使用**key\_len 最小的二级索引**进行扫描。

### count(字段) 执行过程

采用全表扫描的方式来计数



## InnoDB和MyISAM计数的区别

MyISAM 引擎执行 count 函数只需要 **O(1)复杂度**，因为每张 MyISAM 的数据表都有一个 **meta 信息**有存储了**row\_count值**，由**表级锁保证一致性**。

InnoDB 存储引擎是支持事务的，同一个时刻的多个查询，由于多版本并发控制（MVCC）的原因，InnoDB 表“应该返回多少行”也是不确定的，所以无法像 MyISAM一样，只维护一个 row\_count 变量，因此**需要遍历**。

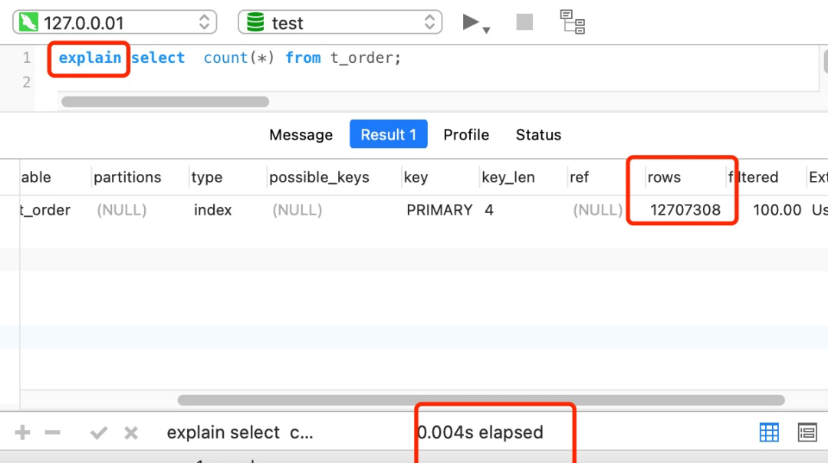
当**带上 where 条件语句之后，MyISAM 跟 InnoDB 就没有区别了**，它们都需要扫描表来进行记录个数的统计。

## 如何优化 count(\*)？

对一张大表进行count(\*)耗时较长

### 第一种，近似值

如果业务对于统计个数不需要很精确，可以使用 **show table status** 或者 **explain** 命令来表进行估算。explain 命令效率很高，因为它并不会真正的去查询：



### 第二种，额外表保存计数值

将计数值保存到单独的一张计数表中。在新增和删除操作时，额外维护这个计数表。