**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

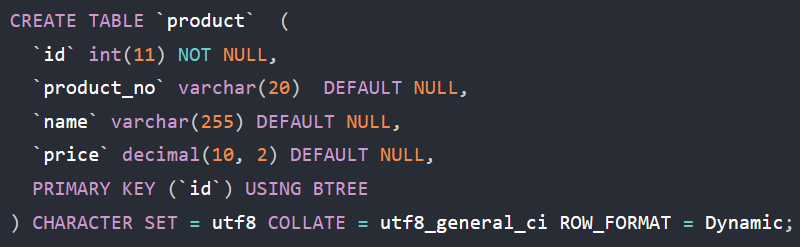
每一种存储引擎支持的索引类型各不相同：

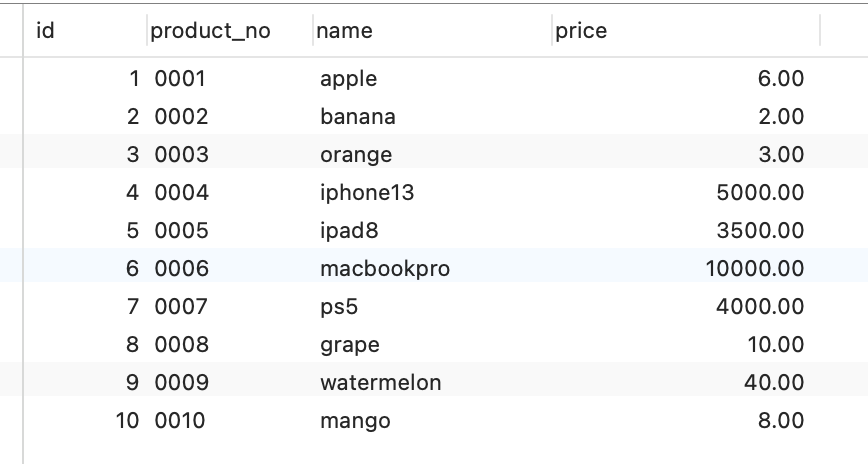


**B+Tree 索引类型是 MySQL 存储引擎采用最多的索引类型，创建的主键索引和二级索引默认使用的是 B+Tree 索引**。

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：

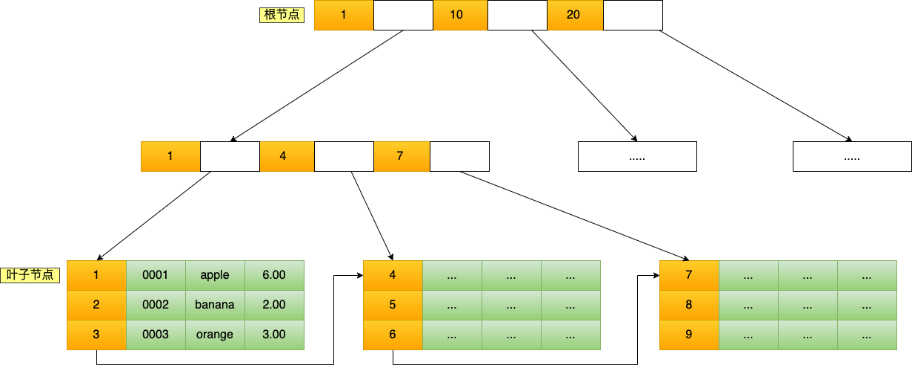




#### 通过主键查询商品数据的过程



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**）：



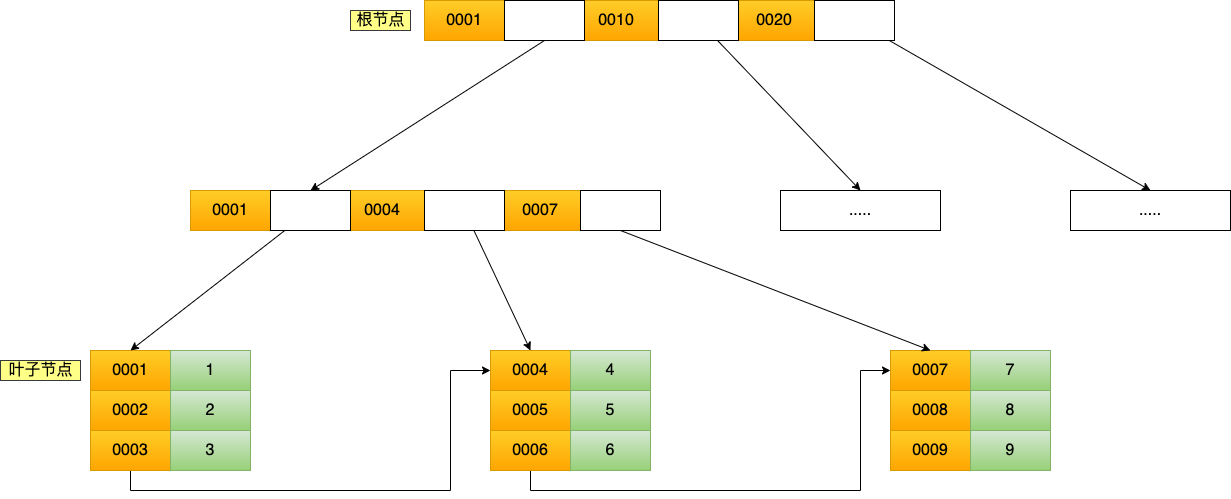
数据库的索引和数据都是存储在硬盘的，**读取一个节点可以看作一次磁盘 I/O 操作**。B+Tree 存储千万级的数据只需要 **3-4 层高度**就可以满足，这意味着从千万级的表查询目标数据最多需要 3-4 次磁盘 I/O，所以**B+Tree 相比于 B 树和二叉树来说，最大的优势在于查询效率很高，因为即使在数据量很大的情况，查询一个数据的磁盘 I/O 依然维持在 3-4次。**

#### 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。

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。

### 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。