

MySQL

3.索引的基本操作

```
show index from t_user;
```

```
mysql> show index from t_user;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
t_user	0	PRIMARY	1	id	A	0	NULL	NULL		BTREE		

1 row in set (0.00 sec)

2. 单列索引(普通索引|单值索引)

--建表时创建

```
create table t_user(id varchar(20) primary key,name varchar(20),key(name));  
'注意:随表一起建立的索引索引名同列名一致'
```

--建表后创建

```
create index nameindex on t_user(name);
```

--删除索引

```
drop index 索引名 on 表名
```

```
mysql> create table t_user(id varchar(20) primary key,name varchar(20),key(name));  
Query OK, 0 rows affected (0.02 sec)  
  
mysql> show index from t_user;  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| t_user | 0 | PRIMARY | 1 | id | A | 0 | NULL | NULL | | BTREE | | |  
| t_user | 1 | name | 1 | name | A | 0 | NULL | NULL | YES | BTREE | | |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
2 rows in set (0.01 sec)
```

3. 唯一索引

--建表时创建

```
create table t_user(id varchar(20) primary key,name varchar(20),unique(name));
```

--建表后创建

```
create unique index nameindex on t_user(name);
```

```
mysql> create table t_user(id varchar(20) primary key,name varchar(20),unique(name));  
Query OK, 0 rows affected (0.01 sec)  
  
mysql> show index from t_user;  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| t_user | 0 | PRIMARY | 1 | id | A | 0 | NULL | NULL | | BTREE | | |  
| t_user | 1 | name | 1 | name | A | 0 | NULL | NULL | YES | BTREE | | |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
2 rows in set (0.00 sec)
```

4. 复合索引

--建表时创建

```
create table t_user(id varchar(20) primary key,name varchar(20),age  
int,key(name,age));
```

--建表后创建

```
create index nameageindex on t_user(name,age);
```

```
mysql> create table t_user(id varchar(20) primary key,name varchar(20),age int,key(name,age));  
Query OK, 0 rows affected (0.02 sec)  
  
mysql> show index from t_user;  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| t_user | 0 | PRIMARY | 1 | id | A | 0 | NULL | NULL | | BTREE | | |  
| t_user | 1 | name | 1 | name | A | 0 | NULL | NULL | YES | BTREE | | |  
| t_user | 1 | name | 2 | age | A | 0 | NULL | NULL | YES | BTREE | | |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
3 rows in set (0.00 sec)
```

4. 索引的底层原理

1. 思考

---建表

```
create table t_emp(id int primary key,name varchar(20),age int);
```

--插入数据

```
insert into t_emp values(5,'d',22);  
insert into t_emp values(6,'d',22);  
insert into t_emp values(7,'e',21);  
insert into t_emp values(1,'a',23);  
insert into t_emp values(2,'b',26);  
insert into t_emp values(3,'c',27);  
insert into t_emp values(4,'a',32);
```

```
insert into t_emp values(8,'f',53);
insert into t_emp values(9,'v',13);
```

--查询

```
select * from t_emp;
```

id	name	age
1	a	23
2	b	26
3	c	27
4	a	32
5	d	22
6	d	22
7	e	21
8	f	53
9	v	13

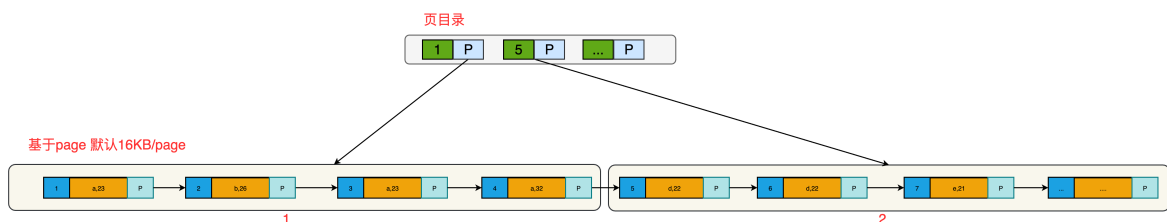
5. 为什么上面数据明明没有按顺序插入,为什么查询时却是有顺序呢?

- 原因是:mysql底层为主键自动创建索引,一定创建索引会进行排序
- 也就是mysql底层真正存储是这样的
- 为什么要排序呢?因为排序之后在查询就相对比较快了 如查询 id=3的我只需要按照顺序找到3就行啦(如果没有排序大海捞针,全靠运气😏!)



6. 为了进一步提高效率mysql索引又进行了优化

- 就是基于页的形式进行管理索引
- 如 查询id=4的 直接先比较页 先去页目录中找,再去 数据目录中找

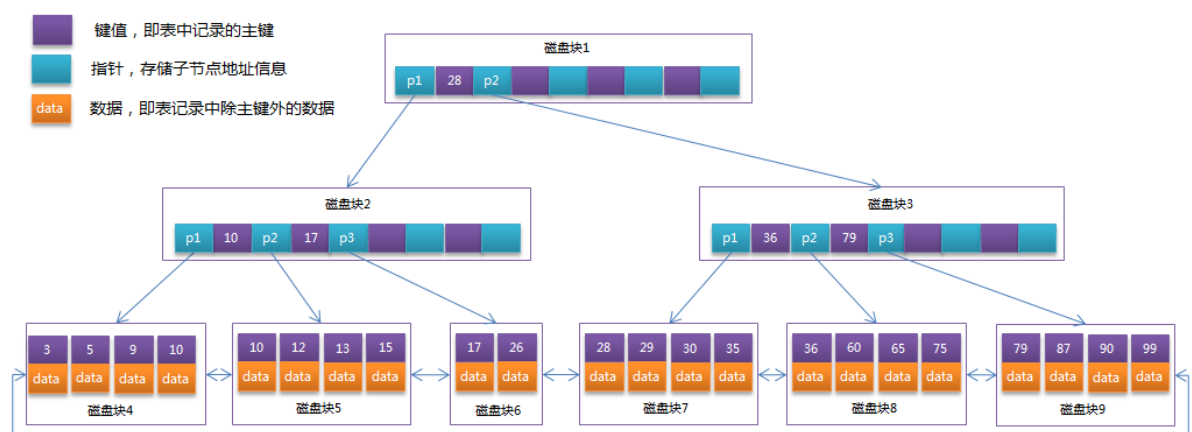


7. 上面这种索引结构称之为B+树数据结构,那么什么是B+树呢?

- 参考资料: <https://www.cnblogs.com/lianzhilei/p/11250589.html>

一般情况下,三到四层的b+树足以支撑千万级别的数据了。

对于索引的选择: 如果varchar小于4用varchar, 大于4用int, 甚至可以只取前几个字符作为索引, 总之要尽量小



B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

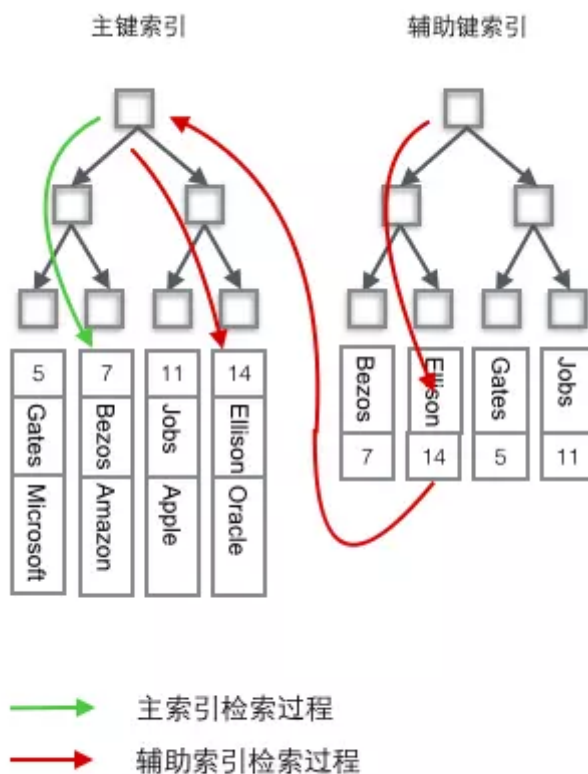
B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
 2. 所有叶子节点之间都有一个链指针。
 3. 数据记录都存放在叶子节点中。
- InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储 $16KB/(8B+8B)=1K$ 个键值（因为是估值，为方便计算，这里的K取值为 $\lfloor 10 \rfloor^3$ ）。也就是说一个深度为3的B+Tree索引可以维护 $10^3 * 10^3 * 10^3 = 10$ 亿条记录。
 - 实际情况中每个节点可能不能填满，因此在数据库中，B+Tree的高度一般都在2~4层。[mysql](#)的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

8. 聚簇索引和非聚簇索引

- 聚簇索引：将数据存储与索引放到了一块，索引结构的叶子节点保存了行数据
- 非聚簇索引：将数据与索引分开存储，索引结构的叶子节点指向了数据对应的位置

注意：在innodb中，在聚簇索引之上创建的索引称之为辅助索引，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引。辅助索引叶子节点存储的不再是行的物理位置，而是主键值，辅助索引访问数据总是需要二次查找。



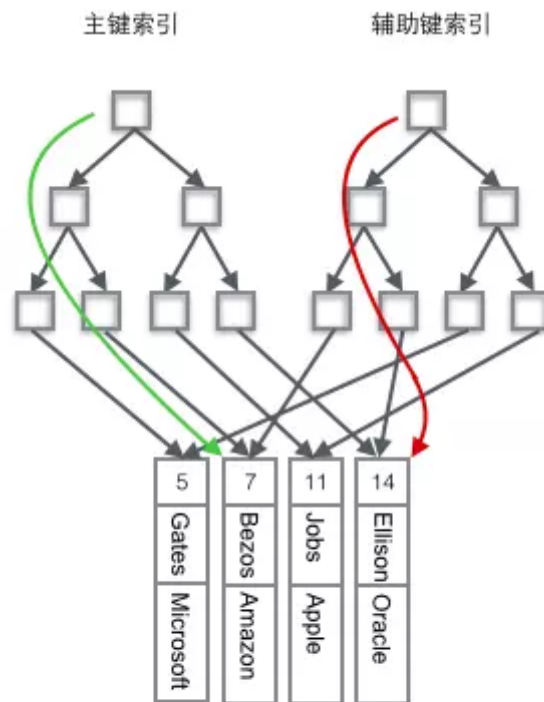
InnoDB（聚簇）表分布

1. InnoDB中

- InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用"where id = 14"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。
- 若对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索引B+树中再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。（重点在于通过其他键需要建立辅助索引）
- **聚簇索引默认是主键**，如果表中没有定义主键，InnoDB 会选择一个**唯一且非空的索引**代替。如果没有这样的索引，InnoDB 会**隐式定义一个主键（类似oracle中的Rowid）**来作为聚簇索引。如果已经设置了主键为聚簇索引又希望再单独设置聚簇索引，必须先删除主键，然后添加我们想要的聚簇索引，最后恢复设置主键即可。

2. MYISAM

- MyISAM使用的是非聚簇索引，**非聚簇索引的两棵B+树看上去没什么不同**，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于**索引树是独立的**，**通过辅助键检索无需访问主键的索引树**。



MyISAM（非聚簇）表分布

9. 使用聚簇索引的优势

- 问题：每次使用辅助索引检索都要经过两次B+树查找，看上去聚簇索引的效率明显要低于非聚簇索引，这不是多此一举吗？聚簇索引的优势在哪？

- 1. 由于行数据和聚簇索引的叶子节点存储在一起，同一页中会有多条行数据，访问同一数据页不同行记录时，已经把页加载到了Buffer中（缓存器），再次访问时，会在内存中完成访问，不必访问磁盘。这样主键和行数据是一起被载入内存的，找到叶子节点就可以立刻将行数据返回了，如果按照主键Id来组织数据，获得数据更快。

- 2. 辅助索引的叶子节点，存储主键值，而不是数据的存放地址。好处是当行数据放生变化时，索引树的节点也需要分裂变化；或者是我们需要查找的数据，在上一次IO读写的缓存中没有，需要发生一次新的IO操作时，可以避免对辅助索引的维护工作，只需要维护聚簇索引树就好了。另一个好处是，因为辅助索引存放的是主键值，减少了辅助索引占用的存储空间大小。

10. 聚簇索引需要注意什么？

- 当使用主键为聚簇索引时，主键最好不要使用uuid，因为uuid的值太过离散，不适合排序且可能出线新增记录的uuid，会插入在索引树中间的位置，导致索引树调整复杂度变大，消耗更多的时间和资源。

- 建议使用int类型的自增，方便排序并且默认会在索引树的末尾增加主键值，对索引树的结构影响最小。而且，主键值占用的存储空间越大，辅助索引中保存的主键值也会跟着变大，占用存储空间，也会影响到IO操作读取到的数据量。

11. 为什么主键通常建议使用自增id

- 聚簇索引的数据的物理存放顺序与索引顺序是一致的，即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的。如果主键不是自增id，那么可以想象，它会干些什么，不断地调整数据的物理地址、分页，当然也有其他一些措施来减少这些操作，但却无法彻底避免。但，如果是自增的，那就简单了，它只需要一页一页地写，索引结构相对紧凑，磁盘碎片少，效率也高。

12. 什么情况下无法利用索引呢？

- 1. 查询语句中使用LIKE关键字

在查询语句中使用 LIKE 关键字进行查询时，如果匹配字符串的第一个字符为“%”，索引不会被使用。如果“%”不是在第一个位置，索引就会被使用。

- 2. 查询语句中使用多列索引

多列索引是在表的多个字段上创建一个索引，只有查询条件中使用了这些字段中的第一个字段，索引才会被使用。

- 3. 查询语句中使用OR关键字

查询语句只有OR关键字时，如果OR前后的两个条件的列都是索引，那么查询中将使用索引。如果OR前后有一个条件的列不是索引，那么查询中将不使用索引。