1. 索引

1.1. 什么是索引

Mysql 官方对于索引的定义是：索引可以帮助 mysql 高效获取数据的数据结构。即索引是数据结构。数据库在执行查询的时候，如何没有索引存在的情况下，会采用全表扫描的方式进行查找。如果存在索引，则会先去索引列表中定位到特定的行或者直接定位到数据，从而可以极大地减少查询的行数，增加查询速度。

可以类比为一部字典开头的目录。

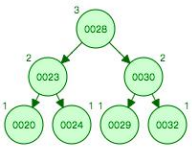
1.2. 索引数据结构

二叉树
红黑树
Hash 表
B 树
常见的数据结构，究竟哪种数据结构更适合在数据库的索引中使用呢?
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

1.2.1. 二叉树（平衡）、红黑树

未加索引，则依次扫描查找

id	name	age
1	zhangsan	32
2	lisi	24
3	wangwu	28
4	zhaoliu	29
5	xiaoming	20
6	hanmeime	23
7	lilei	30



未添加索引之前，如果需要查找某行数据，则需要根据查询条件遍历全表去查找。添加索引之后，查找某行数据，则只需要几次查询即可查到该索引数据，同时二叉树中的每一个元素也保存了相应行数据的磁盘地址，通过该磁盘地址，便可以定位到对应行的数据。但是如果

二叉树作为索引的话会存在什么样的问题？

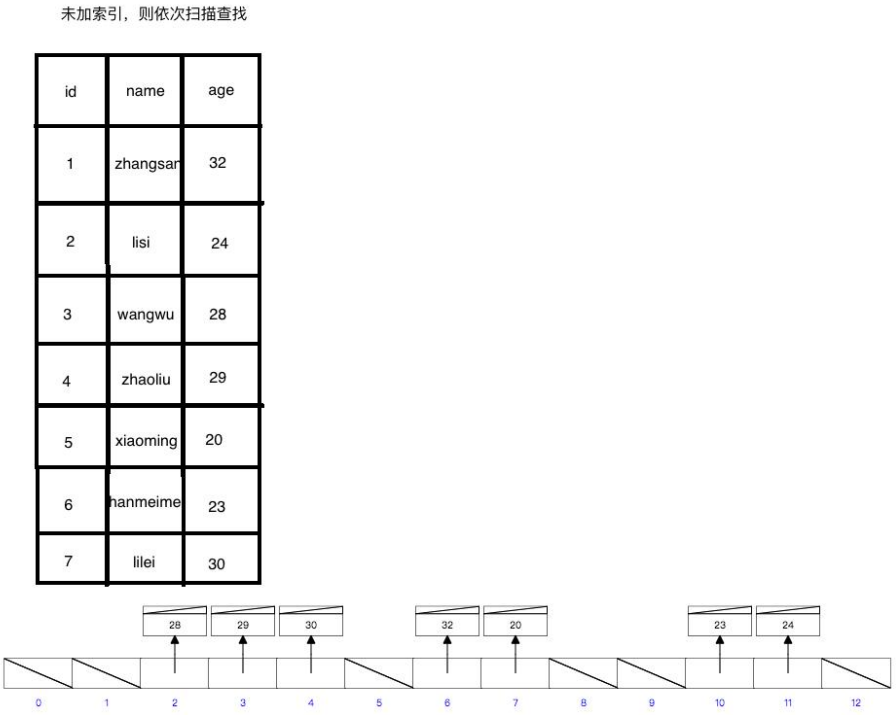
1.2.2. Hash 表

也叫散列表，根据相关的 key 而直接访问的数据结构。做法很简单，把 key 通过一个固定的运算转换成一个数字，然后将这个数字对数组的长度取余，最终的结果就当做数组的下标。对应的数据就放在该下标处。

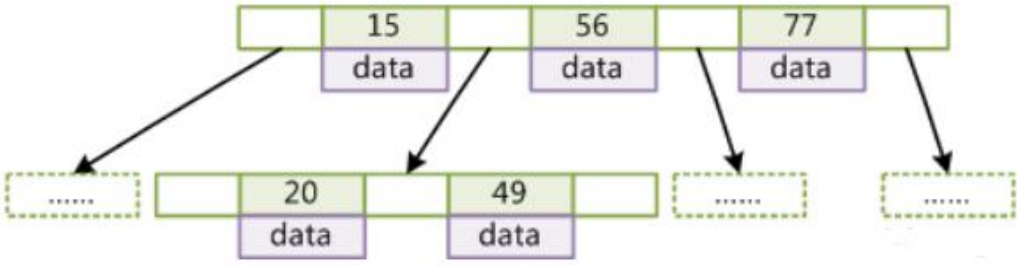
如果 hash 表作为索引，其查询效率也是很高的。但是 hash 会普遍用于数据库的索引上面来吗？

Hash 索引会有以下几种问题：

1. hash 索引仅能够查找=，in 等查找，无法进行范围查找
2. hash 索引无法用来进行排序（经过运算过后的数字大小和原本数字大小没有关系）
3. 如果设置了若干字段的一个组合索引，那么 hash 索引无法利用部分字段进行索引查找，比如设置了用户名、密码、邮箱的联合索引，那么无法使用用户名、密码来通过索引查找。
4. Hash 索引很有可能会导致不同的数据经过运算之后得到相同的 hash 值，因此即便找到了对应的 hash 值所在的下标，仍有可能需要进行再次扫描表数据。
5. 如果存在大量的 hash 值相等的情况，那么 hash 索引此时的查询性能不一定优秀



1.2.3. B 树



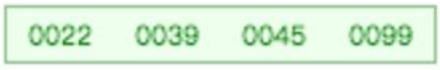
B 树也叫 B-树。是一种多路平衡查找树。B 树的定义如下：
对于一颗 m 阶的 B 树而言（阶数表示一个节点最多有多少个孩子节点）：

- 每个节点最多有 m-1 个 key
- 根节点最少有一个 key，两个子女
- 非根节点包含的 key 个数满足： ⌈m/2⌉ - 1 <= j <= m - 1
- 每个节点中的 key 都是按照从小到大的顺序排列
- 所有叶子节点位于同一层

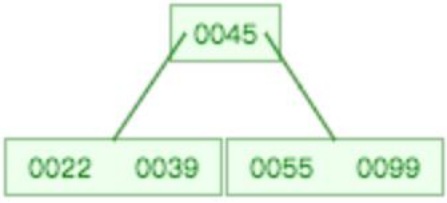
1.2.3.1. B 树插入

对于一颗 5 阶的 B 树，有以下几个特点：
每个节点至多有 5 个孩子子节点
每个节点内至多有 4 个 key，至少 2 个 key
每个节点内有 n 个 key，以及 n + 1 个指针

在 B 树种插入 39，22，99，45

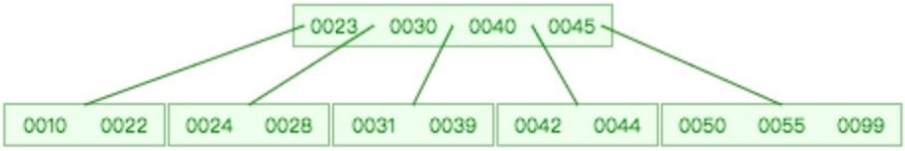


继续插入 55，节点内 key 个数为 5 个，要发生裂变

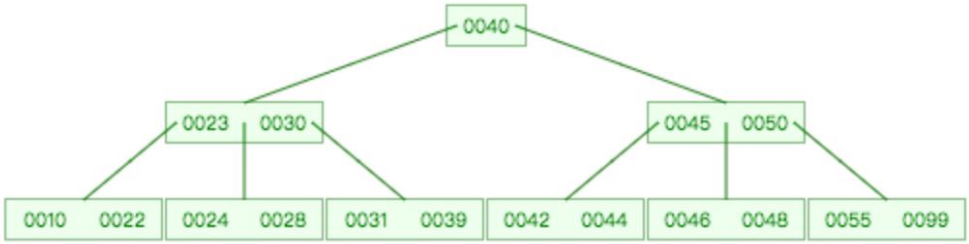


以中间值进行分开，如图所示
继续插入数据 10，23，40，同样左下的节点超过 4 个 key 会发生裂变

继续插入 24，28，30，42，44，50



继续插入 46，48，最右侧会超过 4 个 key，然后发生裂变，中间值向上，进入父节点，但是这个时候，父节点也会超过 4 个 key，所以进一步以中间值向上裂变，形成下图所示



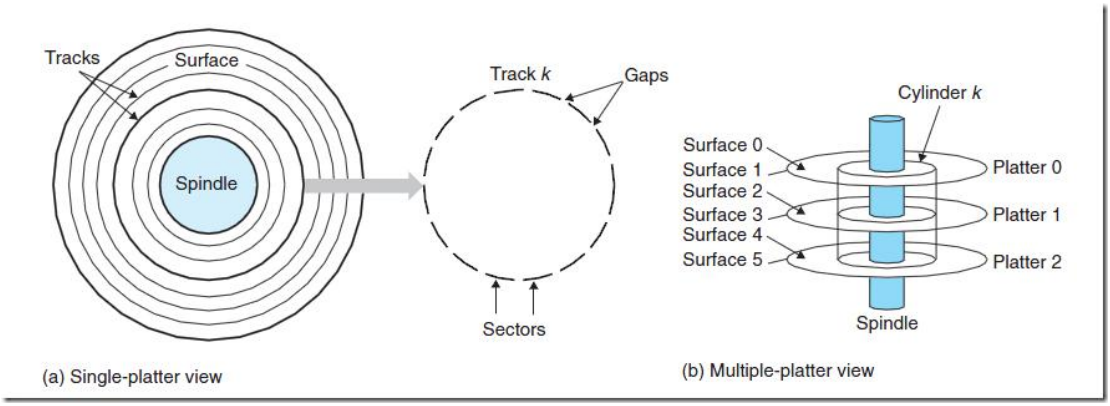
从图示可以看出，如果想查询某个数字，则需要经过三次查询即可查到对应的数据。索引是存在于磁盘中的，也就是说要经过三次磁盘 IO 操作便可定位到对应的索引值。

问题一：B 树相较于平衡二叉树或者红黑树，最大的优势在什么地方？？

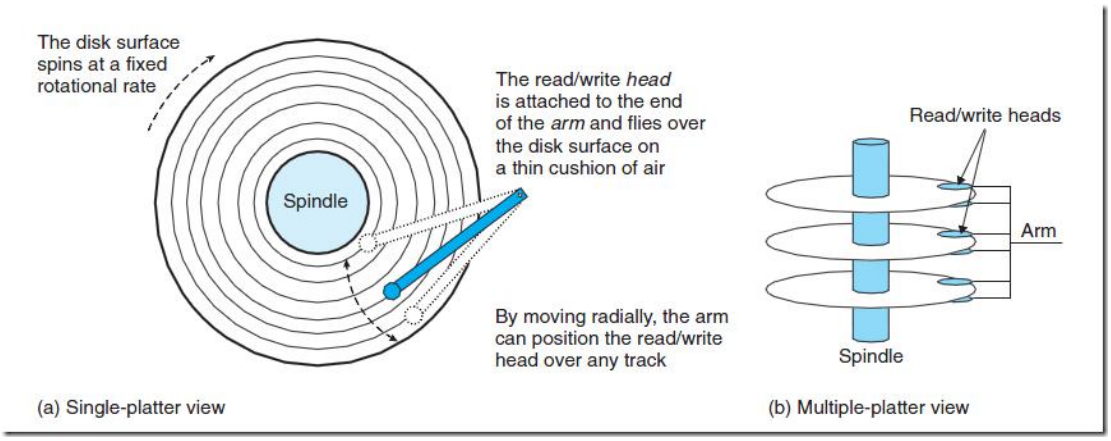
尤其是数据量越多，体现越明显。
对于 B 树而言，如果节点内存储的索引数量越多，那么即便 B 树的高度只有三层或者四层，也可以存储千万条以上的数据。
一般情况下,B 树的高度如果是 3，那么就需要进行三次查询，也就是说需要经过三次磁盘 IO（内存磁盘进行一次交互），查询的限速步骤主要在于磁盘 IO，那么

问题二：如果将千万条数据全部存放在一个节点内，不是只需要一次磁盘 IO 就可以找到对应的数据了吗，为什么不采用这样的方式呢?

首先要清楚磁盘和内存是如何交互的。磁盘的结构如图:



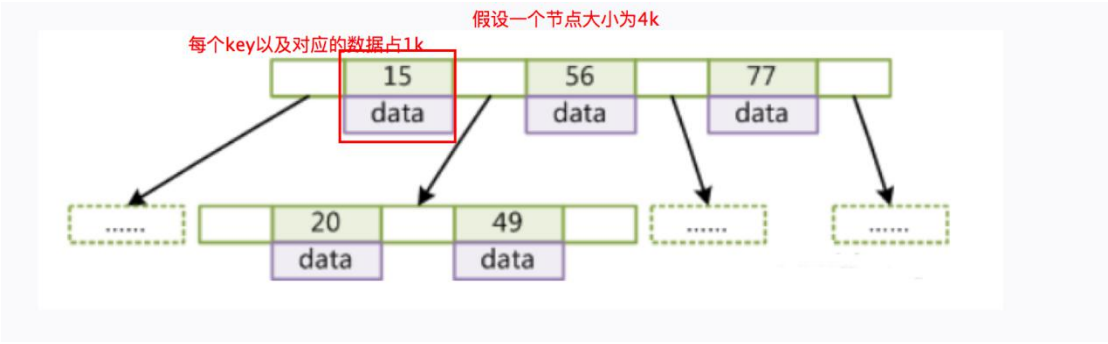
磁盘由盘片组成。盘片有两面，称之为盘面 surface。盘面上覆盖磁性材料。中间是一个可以旋转的轴 Spindle，使得整个盘面能够旋转。通过速率为 5400 转每分钟或者 7200 转每分钟。每个盘面又由一系列的同心圆组成，称之为磁道 track。磁道又被划分为一组组扇区 sector。扇区的大小大概都相等，为 512 字节。



当从磁盘中读取数据时，利用读写头找到对应的磁道，这个步骤称之为寻道。找到磁道后，盘片开始转动，磁道上的每个位都可以被磁头感知到，然后读取到其内容，对磁盘的访问整个过程可以分为寻道时间和访问时间。一般情况下，寻道时间较慢。

由于磁盘存取的速度比内存慢很多，所以磁盘读取时，通常情况下并不是按需读取，而是会预读一部分数据。预读的长度通常情况下为一个页的整数倍。一个页一般情况下大小为 4k。也就是说一次磁盘 IO 通常只会读取 4k 的几倍。因此，把全部数据写入到一个节点中，也并没有太大用处，因为一般只会读取 4k 或者 4k 的几倍。

数据库的实现者也利用磁盘预读的这一点，将一个节点的大小设为 4k 或者 4k 的几倍。最好的状态是一次磁盘 IO 就把一个节点里全部数据给读取到内存中了。



假设一个节点的大小是 4k，然后每个 key 以及对应的数据加起来一共 1k，那么该节点可以存放多少个 key，即索引字段？
那么，如何才能让每个节点存储更多的索引字段呢？数据库底层采用的是 B 树吗？

1.2.4. B+树

数据库底层采用的其实是 B+树来作为索引。它可以看成是 B 树的变种。具有以下特点：

- 非叶子节点不存储 data，只存储 key
- 所有的叶子节点存储完整的一份 key 信息以及 key 对应的 data
- 每一个父节点都出现在子节点中，是子节点的最大或者最小的元素
- 每个叶子节点都有一个指针，指向下一个数据，形成一个链表

特点：B+树由于非叶子节点不存储数据，仅在叶子节点才存储数据，所以，单个非叶子节点可以存储更多的索引字段。
问题三：一个索引树最多能够存储大概多少条数据？
假设一颗索引树的高度为 3，那么一个数据库中一个页的大小是多大呢？比如在 innodb 存储引擎中，页的大小（节点大小）是 16kb，那么如果索引里面存放的是主键的 id 值，比如 bigint 类型，占用 8 个字节，一个指针所占用的空间大概 6 个字节。那么一个节点内大概可以存储多少个索引字段？
1170 个索引字段。如果索引的高度是 3 层，非叶子节点存储的 key 和数据加起来大概 1k 左右，那么最终总条数为 1170*1170*16=2100w 条数据。

问题四：索引为什么使用 B+树，而不使用其他数据结构比如二叉树、红黑树、hash 表、b 树等等？

1.3. 索引的具体实现

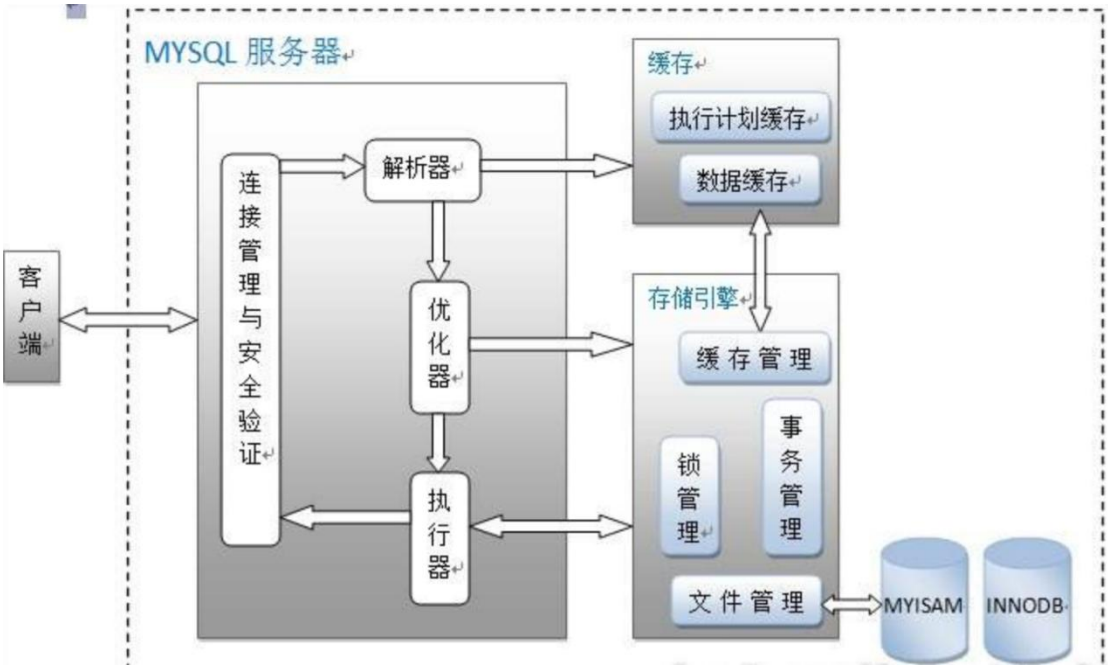
介绍索引具体实现之前，先介绍一下数据库的组成结构。为什么？
因为索引的具体实现和不同的引擎有关。
Mysql 数据库的基本组成架构为：
连接器：负责管理连接，权限的验证等。

解析器：首先 mysql 需要知道你想做什么。因此需要对输入的 sql 进行解析。首先进行词法分析，需要识别出里面的字符串代表什么意思。比如 select 代表查询，T 代表某张表，ID 代表某张表的列字段叫 id；之后进行语法分析，根据语法规则，判断输入的 sql 语句是否符合 mysql 语法。

优化器：经过解析之后，mysql 就知道你需要做什么事情了。但是在真正执行之前还需要经过优化器处理。比如当表中存在多个索引的时候，选择哪个索引来使用。或者多表关联的时候，选择各个表的连接先后顺序。

执行器：开始执行之前首先确认对该表有无执行查询的权限。如果没有，则返回错误的信息提示。如果有权限，则开始执行。首先根据该表的引擎类型，使用这个引擎提供的接口。比

如查询某表，然后利用某字段查找，如果没有添加索引，则调用引擎的接口取出第一行数据，判断结果是不是，如果不是，依次再调用引擎的下一行数据，直至取出这个表中所有的数据。如果该字段有索引，执行过程也大致相似，



所以具体的数据是保存在引擎中的。在 Mysql 中，常见的引擎有 MyISAM 和 InnoDB。

1.3.1. MyISAM 和 InnoDB 区别

1. Innodb 支持事务，MyISAM 不支持事务，对于 Innodb 中的每条 sql 语句都自动封装成事务，自动提交，影响速度
2. Innodb 支持外键，MyISAM 不支持外键
3. Innodb 是**聚集索引**，数据文件和索引绑在一起（数据本身就是按照索引的形式来存储的）。MyISAM 是**非聚集索引**，索引和数据文件是分开的
4. Innodb 不保存表的行数，查询某张表的行数会全表扫描。MyISAM 会保存整个表的行数，执行速度很快
5. Innodb 支持表锁和行锁（默认），而 MyISAM 支持表锁。但是 Innodb 的行锁是通过索引实现的，如果没有命中索引，则依然会使用表锁
   Innodb 表必须要有一个主键（如果用户不设置，那么引擎会自行设定一列当做主键），MyISAM 则可以没有
6. Innodb 的存储文件是 frm 和 ibd，而 MyISAM 是 frm、myd、myi 三个文件。Frm 是表定义文件，ibd 是数据文件；myd 是数据文件、myi 是索引文件

如何选择？

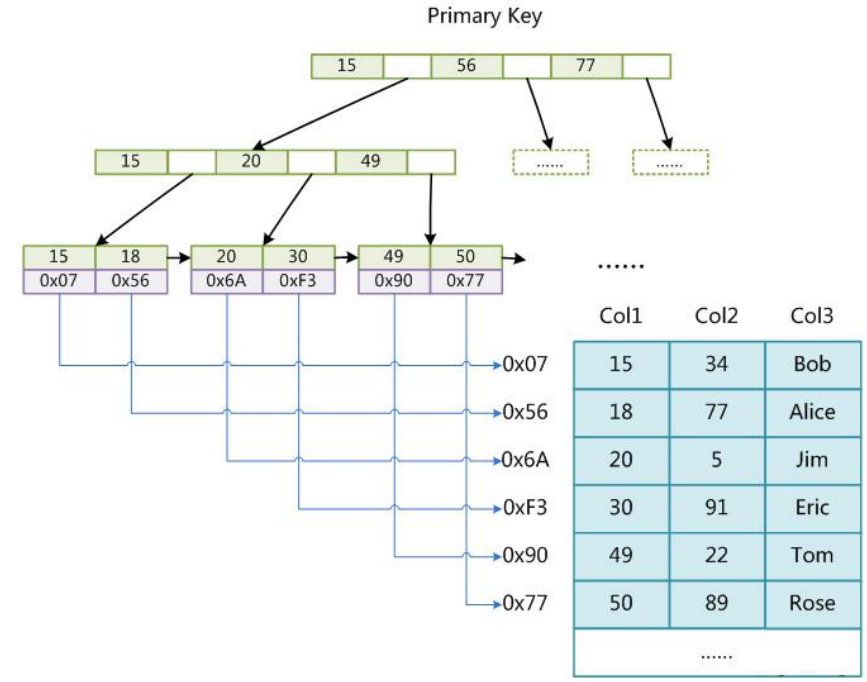
是否需要事务？如果不需要，则可以使用 MyISAM

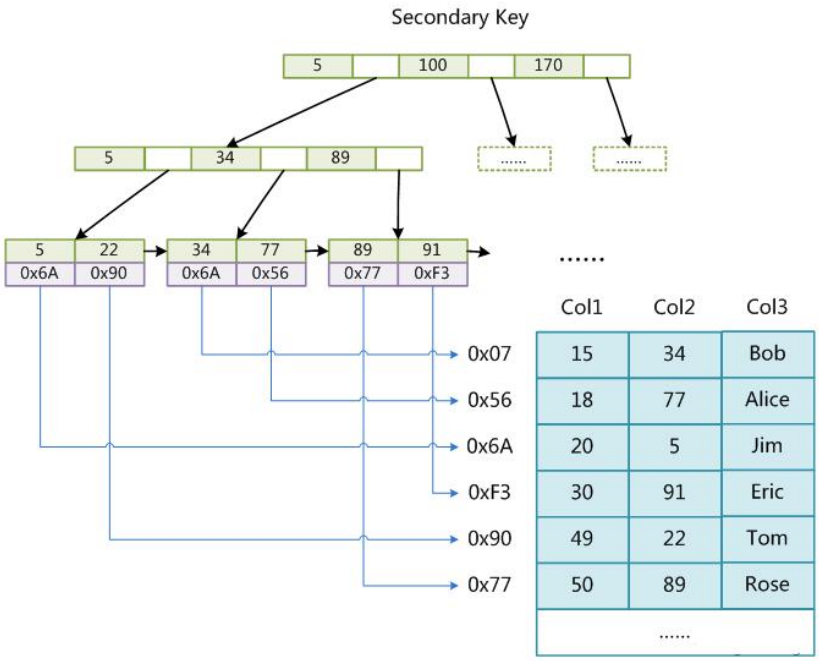
绝大多数操作是否是查询？如果是，可以选择 MyISAM，有读也有写，则选择 Innodb

1.3.2. MyISAM 和 InnoDB 索引实现

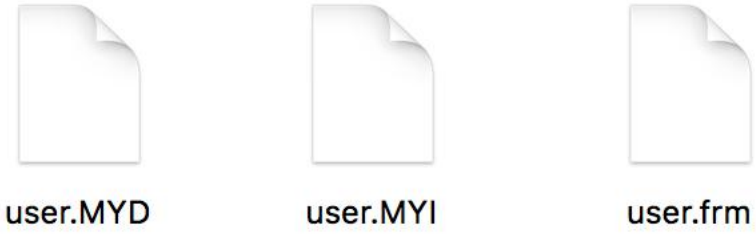
1.3.2.1. MyISAM 索引的实现

MyISAM 的索引是非聚集索引。什么叫非聚集？MyISAM 的索引文件和数据文件是分离的。





主键索引的实现方式和非主键索引（辅助索引）的实现方式并没有太大的区别。MyISAM 的文件有三个文件组成：



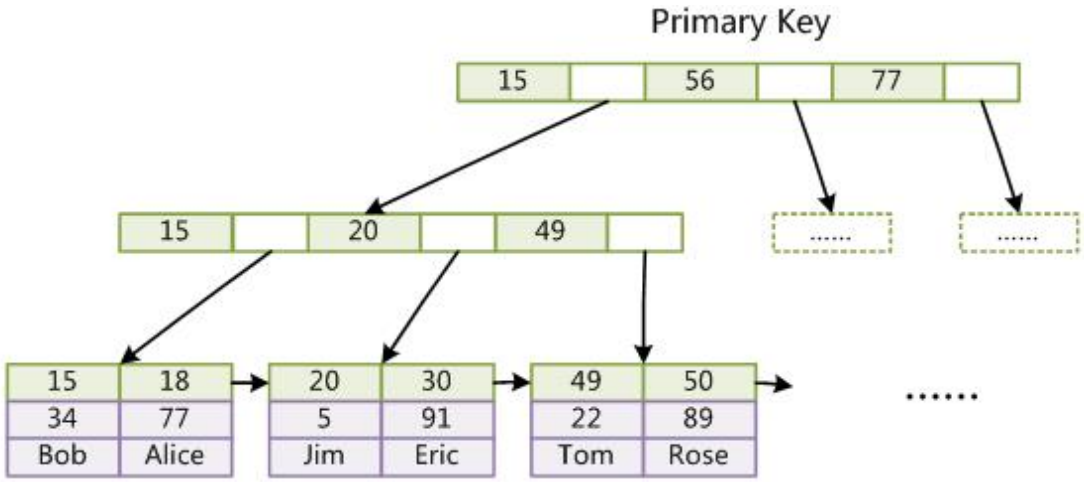
MYI 是索引文件。索引文件中存放的是对应数据的文件指针，接着会去 MYD 文件中去找对应指针的数据。

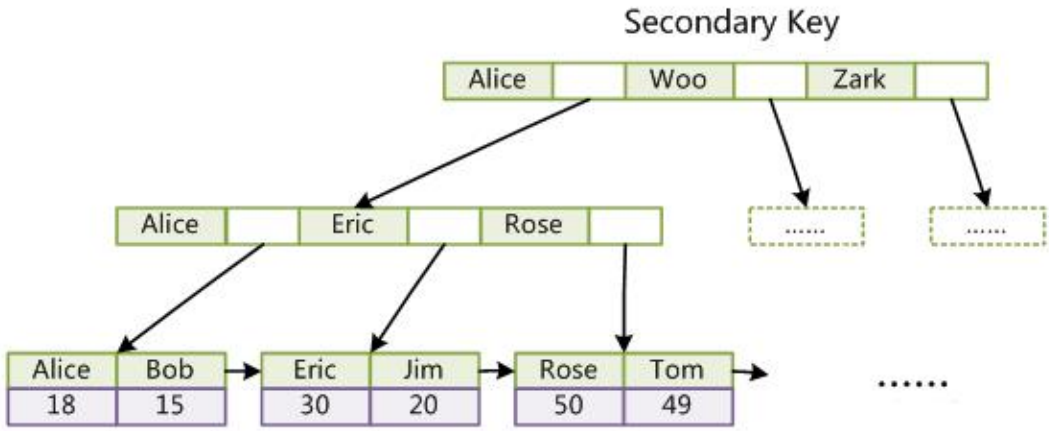
1.3.2.2. Innodb 的索引实现

Innodb 的文件只有一个表结构文件和数据文件。数据文件本身就是索引文件。Innodb 的索引是聚集索引形式。索引和文件数据是存放在一起的。

Innodb 索引的特点：

1. 数据文件本身就是索引文件
2. 数据本身就是按照 B+树索引组织起来的一个索引文件
3. 对于主键索引的 B+树的叶子节点包含了完整的数据信息,对于非主键索引，叶子节点存放的是主键的 id





18 代表什么意思？主键的值。

对于 InnoDB 而言，主键索引：data 里面存放的是完整的数据

非主键索引（二级索引）：data 里面存储的是主键的值

为什么要这么说？

数据的一致性更容易维护。

问题五：为什么 Innodb 表必须要有主键，同时推荐使用自增的整数作为主键？

因为对于 Innodb 表而言，即便不设置主键，那么引擎也会选择一个列或者生成一个隐藏列来当做主键，可能并不是十分好。

使用自增整数作为主键的原因，（如果采用 UUID 来生成一个随机字符当做主键，那么插入的位置是不固定的，可能靠前，也可能靠后，后面再次插入进来，很有可能会导致原先的节点重新分裂。）

问题六：为什么非主键索引叶子节点存放的是主键的值？

1.3.2.3. 联合索引的实现

联合索引指的是对多列创建了索引。比如对 a，b，c 创建了一个联合索引，其实相当于创建了 a 单列索引、（a，b）联合索引以及（a，b，c）联合索引。索引最左前缀原理。

先对 a 进行排序，再排 b，在排 c，不打乱前面的排序规则。

1.4. 索引语法

查看某张表的索引：show index from 表名;

创建普通索引：alter table 表名 add index 索引名(索引列);

创建复合索引：alter table 表名 add index 索引名(索引列 1，索引列 2);

删除某张表的索引：drop index 索引名 on 表名;

1.5. Explain 详解

先创建三张表 student、course、student_course
Insert into student values (null,'zhangsan');
Insert into student values (null,'lisi');
Insert into student values (null,'wangwu');
Insert into student values (null,'zhaoliu');

Insert into course values(null,'java');
Insert into course values(null,'c++');
Insert into course values(null,'python');

Insert into student_course values (null,1,1);
Insert into student_course values (null,2,1);
Insert into student_course values (null,3,1);
Insert into student_course values (null,4,1);

使用 Explain 可以模拟优化器执行 sql 语句，从而知道 Mysql 是如何处理 sql 的。
Explain 中的列

1.5.1. id 列

id 是 select 的编号，有几个 select 就有几个 id 值。在 Mysql 中 select 查询分为简单查询（SIMPLE）和复杂查询（PRIMARY）。复杂查询又分为：简单子查询、派生表（from 语句的子查询）、union 查询。
id 越大表示执行的优先级越高，越先执行；id 相同则依次从上向下执行。

1.5.1.1. 简单查询

```
explain select * from student s,course c,student_course sc where s.sid = sc.sid and c.cid = sc.cid;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	c	(NULL)	ALL	PRIMARY	(NULL)	(NULL)	(NULL)	2	100	(NULL)
1	SIMPLE	sc	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where; Using join buffer (Block Nested Loop)
1	SIMPLE	s	(NULL)	eq_ref	PRIMARY	PRIMARY	4	test1.sc.sid	1	100	(NULL)

1.5.1.2. 复杂查询

1.5.1.2.1. 简单子查询

explain select (select sname from student limit 1) from student_course;

```
explain select (select sname from student limit 1) from student_course;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	student_course	(NULL)	index	(NULL)	PRIMARY	4	(NULL)	6	100	Using index
2	SUBQUERY	student	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	100	(NULL)

1.5.1.2.2. from 语句子查询

explain select s.sname from student s where s.sid = (select sc.sid from student_course sc where sc.cid = (select c.cid from course c where c.cname = 'python') limit 1);

```
explain select s.sname from student s where s.sid = (select sc.sid from student_course sc where sc.cid = (select c.cid from course c where c.cname = 'python') limit 1);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	s	(NULL)	const	PRIMARY	PRIMARY	4	const	1	100	(NULL)
2	SUBQUERY	sc	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where
3	SUBQUERY	c	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2	50	Using where

1.5.1.2.3. union 查询

explain select s.sname,c.cname,sc.* from student s,course c,student_course sc where s.sid = sc.sid and c.cid = sc.cid and s.sname = 'zhangsan' union
select s.sname,c.cname,sc.* from student s,course c,student_course sc where s.sid = sc.sid and c.cid = sc.cid and c.cname = 'python';

```
explain select s.sname,c.cname,sc.* from student s,course c,student_course sc where s.sid = sc.sid and c.cid = sc.cid and s.sname = 'zhangsan' union
select s.sname,c.cname,sc.* from student s,course c,student_course sc where s.sid = sc.sid and c.cid = sc.cid and c.cname = 'python';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	s	(NULL)	ALL	PRIMARY	(NULL)	(NULL)	(NULL)	4	25	Using where
1	PRIMARY	sc	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where; Using join buffer (Block Nested Loop)
1	PRIMARY	c	(NULL)	eq_ref	PRIMARY	PRIMARY	4	test1.sc.cid	1	100	(NULL)
2	UNION	c	(NULL)	ALL	PRIMARY	(NULL)	(NULL)	(NULL)	2	50	Using where
2	UNION	sc	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where; Using join buffer (Block Nested Loop)
2	UNION	s	(NULL)	eq_ref	PRIMARY	PRIMARY	4	test1.sc.sid	1	100	(NULL)
(NULL)	UNION RESULT	<union1,2>	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	Using temporary

1.5.2. select_type 列

1.5.2.1. SIMPLE

简单查询。不包含子查询和 union 等查询

1.5.2.2. PRIMARY

查询中如果包含子查询，那么最外层的查询叫做 PRIMARY。也叫做主查询。查询的主体。

1.5.2.3. SUBQUERY

select 中的子查询

1.5.2.4. UNION

union 中的后面或者第二个 select 语句

1.5.2.5. UNION RESULT

union 的结果

1.5.3. table 列

这一行 explain 正在访问的哪张表。
当有 union 时，UNION RESULT 的 table 列的值为<union1,2>，1 和 2 表示参与 union 的 select 行 id。

1.5.4. type 列

这一列表示关联类型或访问类型，即 MySQL 决定如何查找表中的行，查找数据行记录的大概范围。
依次从最优到最差分别为： const > eq_ref > ref > range > index > ALL
一般来说，得保证查询达到 range 级别，最好达到 ref

1.5.4.1. const

const:当 MySQL 对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于 where 列表中，MySQL 就能将该查询转换为一个常量
explain select * from (select * from student where sid = 1) s1;

```
explain select * from (select * from student where sid = 1) s1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student	(NULL)	const	PRIMARY	PRIMARY	4	const	1	100	(NULL)

1.5.4.2. eq_ref

explain select s.sid from student s, student_course sc where s.sid = sc.sid;
唯一性索引。一般情况下唯一性索引返回的结果为 eq_ref 级别。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sc	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	100	Using where
1	SIMPLE	s	(NULL)	eq_ref	PRIMARY	PRIMARY	4	test1.sc.sid	1	100	Using index

返回的数据具有唯一性。主要用在联表查询的情况。

1.5.4.3. ref

非唯一性索引，对于每个索引键的查询，返回匹配的所有行数据。
比如，在 student 表中插入同名学生
alter table student add index student_index(sname);
explain select * from student s where s.sname = 'zhangsan';

```
alter table student add index student_index(sname);
explain select * from student s where s.sname = 'zhangsan';
```

给 sname 添加索引，通过 sname 查询

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s	(NULL)	ref	student_index	student_index	123	const	2	100	Using index

1.5.4.4. range

检索指定范围的列，where 条件后面是一个范围查询，比如，between、>、<等
explain select * from student where sid between 2 and 5;

```
explain select * from student where sid between 2 and 5;
explain select * from student where sid > 2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student	(NULL)	range	PRIMARY	PRIMARY	4	(NULL)	3	100	Using where

1.5.4.5. index、all

查询全部索引的数据。Index 和 all 的区别在于，index 是遍历所有索引中的数据，而 all 是遍历全部的数据。
explain select sid from student;

```
explain select sid from student;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student	(NULL)	index	(NULL)	student_index	123	(NULL)	5	100	Using index

explain select cname from course；查询结果 type 为 all，表示该查询语句性能很低，这个时候可以通过添加索引，将其优先级向前提。

```
explain select cname from course;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	course	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2	100	(NULL)

1.5.5. possible_keys

显示可能使用哪些索引来查找。Explain 出现 possible_keys 有列，而 key 显示 null 情况，因为表数据不多，mysql 认为索引对此查询帮助不大，选择全表扫描。
例如，student 表的 sid 和 sname，给 sname 添加 index，sid 为主键，那么该表的所有字段均有索引，执行 select sid,sname from student;

```
explain select sid,sname from student;
```

结果 1 | 剖析 | 状态

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1 SIMPLE	student	(Null)	index	(Null)	student_index	93	(Null)	4	100.00	Using index

1.5.6. key 列

该列显示了 mysql 实际选择哪个索引来优化该表的访问。

1.5.7. key_len 列

这一列显示 mysql 在索引里使用的字节数，通过这个值可以算出具体使用了索引中的哪些列。
key_len 计算规则：

1. 字符串
   a) char(n)：n 字节长度
   b) varchar(n):如果是 utf-8，则长度 3n + 2
2. 数值类型
   a) Tinyint：1 字节

b) Smallint：2 字节
c) Int:4 字节
d) Bigint：8 字节

3. 时间类型
   a) Date:3 字节
   b) Timestamp：4 字节
   c) Datetime：8 字节
4. **如果该字段允许 null，则还需要 1 字节来记录是否为 null**

如果你新建了一个（a,b,c）三列的复合索引，那么其实是相当于新建了 a 和 a,b 以及 a,b,c 三个索引。**最左前缀原则**。

1.5.8. ref 列

注意与 type 中 ref 的区分。两者不是一个意思。表示当前表在利用 Key 列记录中的索引进行查询时所用到的列或常量。一般用法为：const 或者另外表的一个字段。
explain select s.sid from student s, student_course sc where s.sid = sc.sid;

```
explain select s.sid from student s, student_course sc where s.sid = sc.sid;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sc	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	100	Using where
1	SIMPLE	s	(NULL)	eq_ref	PRIMARY	PRIMARY	4	test1.sc.sid	1	100	Using index

常量 const
explain select * from student s where s.sname = 'zhangsan';

```
explain select * from student s where s.sname = 'zhangsan';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s	(NULL)	ref	student_index	student_index	93	const	1	100	Using index

1.5.9. rows 列

表示 mysql 通过索引的统计信息，预算出来的所需要读取的行数，但是需要注意的是 rows 的值仅仅是统计学上的一个数据，并不是十分的准确。

1.5.10. Extra 列

这一列展示的是额外信息。常见的有如下：

1.5.10.1. Using index(比较好)

- `Using index` (JSON property: `using_index`)

  The column information is retrieved from the table using only information in the index tree without having to do an additional seek to read the actual row. This strategy can be used when the query uses only columns that are part of a single index.

  For `InnoDB` tables that have a user-defined clustered index, that index can be used even when `Using index` is absent from the `Extra` column. This is the case if `type` is `index` and `key` is `PRIMARY`.

意思是索引树中就可以获取全部的数据，不需要再额外去进行第二次操作。当查询的列是索引的部分列时（比如索引是一个复合索引，查询的是索引列的几个列）。无需回表查询。

覆盖索引：select 查询的数据，从索引中就可以全部获取，无需回表查询。

explain select sid,sname from student;无需回表查询。

```
create table test_order(
    id int primary key auto_increment,
    user_id int,
    order_id int,
    order_status tinyint,
    create_date datetime
);
insert into test_order values(null,1,1,1,now());
insert into test_order values(null,2,2,1,now());
insert into test_order values(null,3,3,1,now());
insert into test_order values(null,3,4,0,now());

create index index_userid_orderid_date on test_order(user_id,order_id,create_date);
```

```
create table test_order
(
    id int auto_increment primary key,
    user_id int,
    order_id int,
    order_status tinyint,
    create_date datetime
);
insert into test_order values (null,1,1,1,now());
insert into test_order values (null,2,2,1,now());
insert into test_order values (null,3,3,1,now());
insert into test_order values (null,3,4,0,now());

explain select user_id,order_id,create_date from test_order;
explain select user_id,order_id,create_date from test_order where user_id = 1;
explain select id from test_order where id = 1;

create index idx_userid_order_id_createdate on test_order(user_id,order_id,create_date);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_order	(NULL)	index	(NULL)	idx_userid_order_id_createdate	16	(NULL)	4	100	Using index

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_order	(NULL)	ref	idx_userid_order_id_createdate	idx_userid_order_id_createdate	5	const	1	100	Using index

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_order	(NULL)	const	PRIMARY	PRIMARY	4	const	1	100	Using index

1.5.10.2. Using where

意味着全表扫描，或者即便使用索引查找的情况下，仍然有部分查询条件不在索引字段中。
Server 对搜索引擎返回的数据所做的过滤。
查询的列未被索引覆盖，where 筛选条件非索引的前导列。
查询的列被索引覆盖，但是搜索条件不被索引覆盖。
查询的列未被索引覆盖，搜索条件也没被索引覆盖。

```
#如下三种写法都是使用Using where
#查询的列未被索引覆盖，但是搜索条件被索引覆盖且不是前导列
explain select order_status from test_order where create_date = '2019-10-22 15:52:24';
#查询的列被索引覆盖，但是搜索条件不被索引覆盖
explain select user_id from test_order where order_status = 1;
#查询的列未被索引覆盖，且搜索条件也没被索引覆盖
explain select * from test_order where order_status = 1;
```

```
explain select user_id,order_id,order_status,create_date from test_order where order_id = 1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_order	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25	Using where

1.5.10.3. Using where;Using index

查询的列被索引覆盖，where 筛选条件是索引列之一但不是索引的前导列。
表示搜索引擎通过索引检索将结果返回，然后在 server 层再通过 where 语句对检索结果进行过滤。
explain select user_id,order_id,create_date from test_order where create_date = '2019-12-02 17:00:00';

explain select user_id from test_order where user_id >= 1 and user_id <= 3;

explain select user_id,order_id,create_date from test_order where user_id = 1 and create_date = '2019-12-02 17:00:00';

```
explain select user_id,order_id,create_date from test_order where order_id = 1 and create_date = '2019-10-22 15:52:24';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_order	(NULL)	index	(NULL)	idx_userid_order_id_createdate	16	(NULL)	4	25	Using where; Using index

Using where 和 using where;using index 区别：
出现 using where 表示 server 拿到引擎处理的数据之后需要进行进一步筛选。
至于有没有 using index，那么就需要看数据是否被索引搜索。

1.5.10.4. Null

没有从索引中取数据，但是从索引中查数据。
explain select * from test_order where user_id = 1

信息 结果 1 剖析 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_order	(Null)	ref	index_userid_orderid_dat	index_userid_orderid_dat	5	const	1	100.00	(Null)

1.5.10.5. Using index condition

Suppose that a table contains information about people and their addresses and that the table has an index defined as `INDEX (zipcode, lastname, firstname)`. If we know a person's `zipcode` value but are not sure about the last name, we can search like this:

```
SELECT * FROM people
  WHERE zipcode='95054'
  AND lastname LIKE '%etrunia%'
  AND address LIKE '%Main Street%';
```

MySQL can use the index to scan through people with `zipcode='95054'`. The second part (`lastname LIKE '%etrunia%'`) cannot be used to limit the number of rows that must be scanned, so without Index Condition Pushdown, this query must retrieve full table rows for all people who have `zipcode='95054'`.

With Index Condition Pushdown, MySQL checks the `lastname LIKE '%etrunia%'` part before reading the full table row. This avoids reading full rows corresponding to index tuples that match the `zipcode` condition but not the `lastname` condition.

SET optimizer_switch = 'index_condition_pushdown=off';
SET optimizer_switch = 'index_condition_pushdown=on';

explain select * from test_order where user_id = 1 and create_date = '2019-12-02 17:00:00';

Using index condition 和 Using index;Using where 很相似。取出的结果中如果索引列全覆盖，那么这个时候是后者；如果是取出的数据包含索引之外的列，那么为前者。

1.5.10.6. Using temporary

Mysql 需要创建一个临时表来处理查询。一般是要进行索引优化。

```
explain select distinct cname from course;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	course	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2	100	Using temporary

```
alter table course add index course_index(cname);
```

之后再次进行查询，显示

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	course	(NULL)	index	course_index	course_index	123	(NULL)	2	100	Using index

1.5.10.7. Using filesort

出现这个信息时，表示 mysql 需要进行一次额外的排序才能够显示出结果。一般出现在 order by 相关 sql 语句中。Filesort 并不是说文件排序，而是说一种排序算法。会在内存中将所有的数据取出来进行排序，之后显示出结果。

名	类型	长度	小数点	不是 null	虚拟	键
id	int	11	0	☑	☐	🔑1
name	varchar	30	0	☐	☐	
english	double	0	0	☐	☐	
math	double	0	0	☐	☐	
chinese	double	0	0	☐	☐	
total	double	0	0	☐	☐	

```
explain select name,total from student_score order by total;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student_score	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	3	100	Using filesort

可以通过给需要排序的字段添加索引来优化

```
alter table student_score add index score_index(total,`name`);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	student_score	(NULL)	index	(NULL)	score_index	132	(NULL)	3	100	Using index

	查询条件索引覆盖	查询条件索引未覆盖	查询条件索引部分覆盖
搜索条件索引覆盖	Using index	Null	Null
搜索条件索引未覆盖	Using where	Using where	Using where
搜索条件索引部分覆盖	Using where;Using index	Using where	Using where(其中搜索条件为索引前导列+非索引列) Using index condition(索引前导列+间隔索引列)

1.6. 索引实践

```
CREATE TABLE `user` (
    `id` bigint(20)  PRIMARY key AUTO_INCREMENT,
    `username` varchar(50) DEFAULT NULL,
    `email` varchar(30) DEFAULT NULL,
    `password` varchar(32) DEFAULT NULL,
    `status`  tinyint(1) NULL DEFAULT 0
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Java 语言中的一个函数、方法。可以反复调用。

```
drop procedure myproc;

create procedure myproc()
begin
    declare num int;
    set num=1;
    while num <= 10000000 do
        insert into `user`(username,email,password) values(CONCAT('username_',num), CONCAT(num ,'@qq.com'), MD5(num));
        set num=num+1;
    end while;
end

call myproc();

alter table user engine = Innodb;
```

新建一张 user 表来测试，给该表添加索引

```
mysql> alter table user add index username_password_email(username,password,email);
Query OK, 0 rows affected (18.39 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

1.6.1. 索引全覆盖

如果设置了一个复合索引，那么最理想的状态就是索引的全部字段都会使用到。

1.explain select * from user where username = 'user_999';

```
mysql> explain select * from user where username = 'username_999999';
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys          | key                    | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | user  | NULL       | ref  | username_password_email | username_password_email | 153     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

2. explain select * from user where username = 'user_999' and password = 'b706835de79a2b4e80506f582af3676a';

```
mysql> explain select * from user where username = 'username_999999' and password = '388ec3e3fa4983032b4f3e7d8fcb65ad';
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys          | key                    | key_len | ref         | rows | filtered | Extra |
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------------+------+----------+-------+
|  1 | SIMPLE      | user  | NULL       | ref  | username_password_email | username_password_email | 252     | const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

3.explain select * from user where username = 'user_999' and password = 'b706835de79a2b4e80506f582af3676a' and email = '999@qq.com';

```
mysql> explain select * from user where username = 'username_999999' and password = '388ec3e3fa4983032b4f3e7d8fcb65ad' and email = '999999@qq.com';
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys          | key                    | key_len | ref               | rows | filtered | Extra |
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------------------+------+----------+-------+
|  1 | SIMPLE      | user  | NULL       | ref  | username_password_email | username_password_email | 345     | const,const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+------------------------+------------------------+---------+-------------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

4. explain select * from user where password = '388ec3e3fa4983032b4f3e7d8fcb65ad' and email = '999999@qq.com'; 不会使用索引

```
mysql>  explain select * from user where password = '388ec3e3fa4983032b4f3e7d8fcb65ad' and email = '999999@qq.com';
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9702644 |     1.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

5. explain select * from user where email = '999999@qq.com';

```
mysql> explain select * from user where email = '999999@qq.com';
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9702644 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

1.6.2. 索引最左前缀原则

如果设置了联合索引，那么在使用的时候，从索引的最左开始，中间尽量不要跨列，否则会导致索引的失效。

1.6.3. 不要对索引的列做任何操作，否则会索引失效

explain select * from user where left(username,5) = 'username_9999';

```
mysql> explain select * from user where username = 'username_9999';
+----+-------------+-------+------------+------+---------------------------+---------------------------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys             | key                       | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------------------+---------------------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | user  | NULL       | ref  | username_password_email | username_password_email | 153     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------------------+---------------------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from user where left(username,5) = 'username_9999';
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9702644 |   100.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

1.6.4. 存储引擎不能使用索引中范围条件右侧的列

若中间索引使用了范围，则后面的索引全失效。

```
mysql> explain select * from user where username = 'username_9999' and password < '388ec3e3fa4983032b4f3e7d8fcb65ad' and email = '9999@qq.com';
+----+-------------+-------+------------+-------+---------------------------+---------------------------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys             | key                       | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------------------+---------------------------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | user  | NULL       | range | username_password_email | username_password_email | 252     | NULL |    1 |    10.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------------------+---------------------------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

```
mysql> desc user;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | bigint(20)  | NO   | PRI | NULL    | auto_increment |
| username | varchar(50) | YES  | MUL | NULL    |                |
| email    | varchar(30) | YES  |     | NULL    |                |
| password | varchar(32) | YES  |     | NULL    |                |
| status   | tinyint(1)  | YES  |     | 0       |                |
+----------+-------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

```
mysql> explain select * from user where username = 'username_9999' and password > '388ec3e3fa4983032b4f3e7d8fcb65ad' and email = '9999@qq.com';
+----+-------------+-------+------------+-------+---------------------------+---------------------------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys             | key                       | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------------------+---------------------------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | user  | NULL       | range | username_password_email | username_password_email | 252     | NULL |    1 |    10.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------------------+---------------------------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

1.6.5. 尽量使用覆盖索引，而减少使用 select *

什么叫做覆盖索引，意思是 select 查询的信息或者说字段已经出现在索引中，无需额外再去回表去查询，这样叫做覆盖索引。可以极大得提升查询的效率。在实际查询过程中，应当极力避免出现 select *的情形。

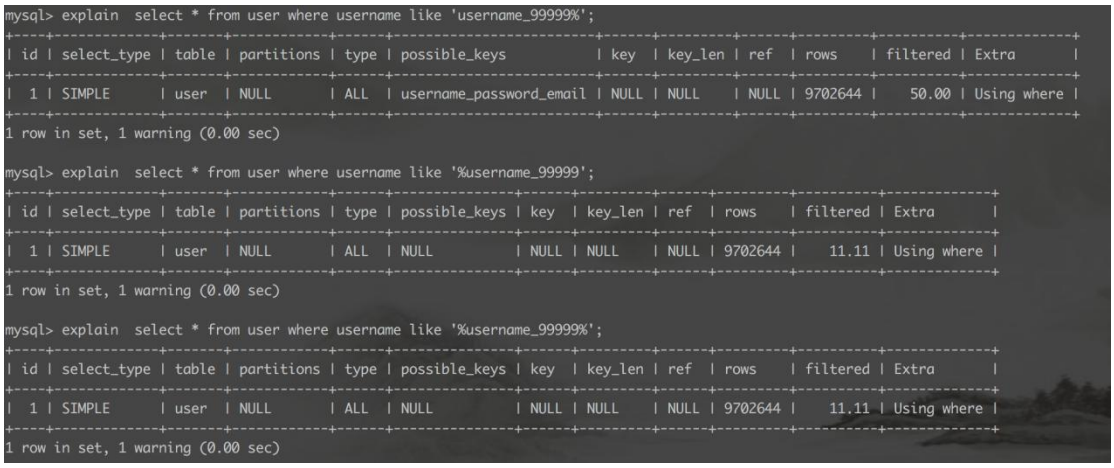
1.6.6. 使用！= 或者<>时会索引失效

而转向全表扫描查询。

explain select * from user where username != 'username_9999';

```
mysql> explain select * from user where username != 'username_9999';
+----+-------------+-------+------------+------+---------------------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys             | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | username_password_email | NULL | NULL    | NULL | 9702644 |    50.01 | Using where |
+----+-------------+-------+------------+------+---------------------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

1.6.7. Like 模糊查询索引字段会失效



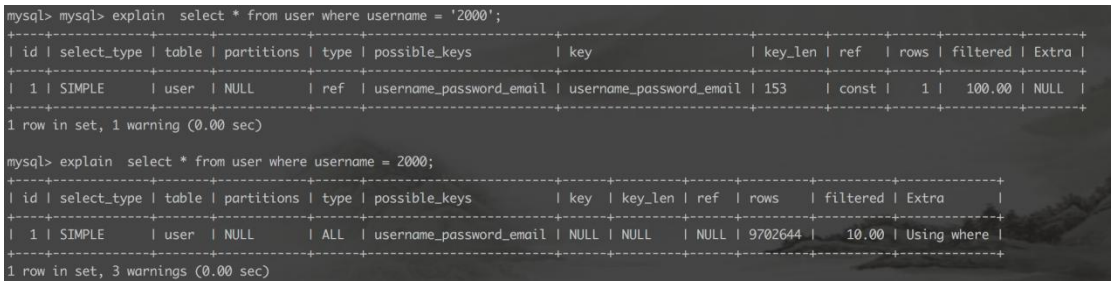
```
mysql> explain  select * from user where username like 'username_99999%';
+----+-------------+-------+------------+------+--------------------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys            | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+--------------------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | username_password_email | NULL | NULL    | NULL | 9702644 |    50.00 | Using where |
+----+-------------+-------+------------+------+--------------------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from user where username like '%username_99999';
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9702644 |    11.11 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from user where username like '%username_99999%';
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9702644 |    11.11 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

如果需要使用%%形式搜索数据，那么当数据量变的非常大时，即便采用覆盖索引也不会有很大的提升，这个时候可以考虑使用搜索引擎处理。ElasticSearch

1.6.8. 字符串如果不加单引号会导致索引失效



```
mysql> mysql> explain  select * from user where username = '2000';
+----+-------------+-------+------------+------+--------------------------+--------------------------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys            | key                      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+--------------------------+--------------------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | user  | NULL       | ref  | username_password_email | username_password_email | 153     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+--------------------------+--------------------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> explain  select * from user where username = 2000;
+----+-------------+-------+------------+------+--------------------------+------+---------+------+---------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys            | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------+------------+------+--------------------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | username_password_email | NULL | NULL    | NULL | 9702644 |    10.00 | Using where |
+----+-------------+-------+------------+------+--------------------------+------+---------+------+---------+----------+-------------+
1 row in set, 3 warnings (0.00 sec)
```