mysql学习

1.索引基本知识点

1.1 什么是索引

索引是帮助 MySQL 高效获取数据的**排好序**的**数据结构**。所以，可以得出：**索引是数据结构**

索引用的数据结构为B+树：

非叶子节点只存储索引信息（不存储索引外的数据信息）

叶子节点存储所有数据信息，且叶子节点间以双向循环链表连接

目的：进一步降低树的高度 + 非叶子结点存放索引的数量 + 叶子结点间双向循环链表提高区间访问的性能，方便范围查找数据。

1.2 主键索引和非主键索引

主键索引：聚簇索引

非主键索引：非聚簇索引

聚簇索引：将数据存储和索引放在了一块，索引结构的叶子节点保存了行数据

非聚簇索引（辅助索引）：将数据和索引分开存储，索引结构的叶子节点指向了数据对应的位置， 叶子节点存储的实际上是主键值，辅助索引访问数据一般需要二次查找（回表查询）

```
select * from tablename where name = "Ellison";
```

换句话说，索引的叶节点中除了存储了索引定义中的字段外,还存储了primary key,从而可以找到对应的行记录,这样才能访问索引外的字段，才能实现回表查询

1.3.索引的重要性

索引对于MySQL数据库的重要性是不言而喻的:

因为缺乏合适的索引,一个稍大的表全表扫描,稍微来些并发,就可能导致DB响应时间急剧飙升,甚至导致DB性能的雪崩;

现在大家普遍使用的Innodb引擎的锁机制依赖于索引,缺乏适合的索引,会导致锁范围的扩大,甚至导致锁表的效果,严重影响业务SQL的并行执行,影响业务的可伸缩性,只有在合适的索引条件下,才是行锁的效果.（没有合理的索引，修改数据的时候，只能使用全表扫描的方案时，执行修改的过程中，就是锁定全表的所有数据，严重影响业务SQL的并行执行，很容易导致行锁争用，连接打满）

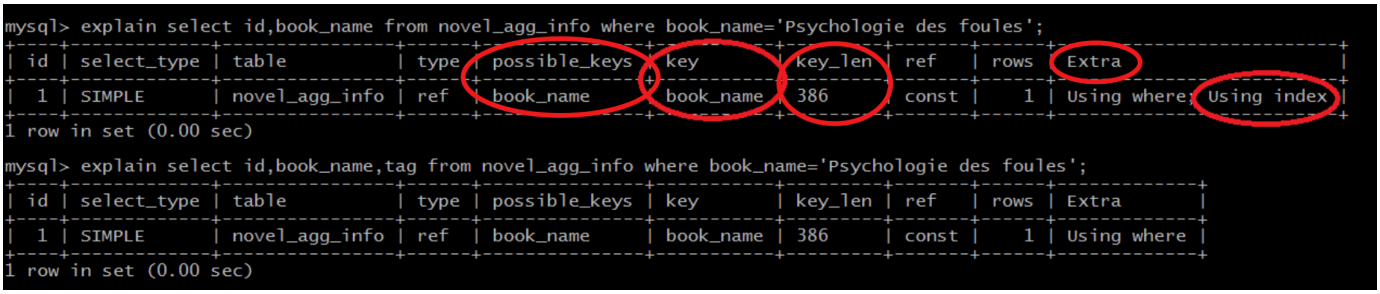
今天我们主要来探讨如何正确使用索引，暂时不去深究索引的底层原理。

2.SQL语句执行计划

索引的目的是为了加快我们SQL查询的速度，那么我们设置了索引后，如何知道我们的索引是否生效了呢?

可以用explain语句，explain查询mysql语句的执行计划

2.1.explain查询mysql语句的输出



查询计划输出有几个关键字段:

possible_keys: 表示查询时，可能使用的索引

key: 查询时实际用到的索引

key_len:索引字段的长度,复合索引用到的索引字段

rows:扫描出的行数(估算的行数)

Extra:执行情况的描述和说明

2.2.无索引情况下

全表扫描，检索行数

前提:

novel_agg_info表数据总量429w+

dir_id无索引

```
mysql> explain select * from novel_agg_info where dir_id = 13301689388199959972;
+----+-------------+----------------+------+---------------+------+---------+------+---------+-------------+
| id | select_type | table          | type | possible_keys | key  | key_len | ref  | rows    | Extra       |
+----+-------------+----------------+------+---------------+------+---------+------+---------+-------------+
|  1 | SIMPLE      | novel_agg_info | ALL  | NULL          | NULL | NULL    | NULL | 4290581 | Using where |
+----+-------------+----------------+------+---------------+------+---------+------+---------+-------------+
1 row in set (0.00 sec)
```

2.3.using_index，是否需要回表访问

普通索引`book_name` 主键索引id

```
mysql> explain select id,book_name from novel_agg_info where book_name='Psychologie des foules';
+----+-------------+----------------+------+---------------+-----------+---------+-------+------+--------------------------+
| id | select_type | table          | type | possible_keys | key       | key_len | ref   | rows | Extra                    |
+----+-------------+----------------+------+---------------+-----------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | novel_agg_info | ref  | book_name     | book_name | 386     | const |    1 | Using where; Using index |
+----+-------------+----------------+------+---------------+-----------+---------+-------+------+--------------------------+
1 row in set (0.00 sec)

mysql> explain select id,book_name,tag from novel_agg_info where book_name='Psychologie des foules';
+----+-------------+----------------+------+---------------+-----------+---------+-------+------+-------------+
| id | select_type | table          | type | possible_keys | key       | key_len | ref   | rows | Extra       |
+----+-------------+----------------+------+---------------+-----------+---------+-------+------+-------------+
|  1 | SIMPLE      | novel_agg_info | ref  | book_name     | book_name | 386     | const |    1 | Using where |
+----+-------------+----------------+------+---------------+-----------+---------+-------+------+-------------+
1 row in set (0.00 sec)
```

两个mysql执行计划都用到了索引，不同的是Extra字段的不同

第一条语句中 Extra字段中有 Using index,第二条语句中没有，

第1个SQL中只需要检索id,book_name字段,这在KEY `book_name` (`book_name`)中都存在了(索引叶节点中都会存储PRIMARY KEY字段ID),不需要回访表去获取其它字段了,Using index即代表这个含义;而第2个SQL中还需要检索tag字段,这在KEY `book_name` (`book_name`)中并不存在,就需要回访表会获取这个字段内容,所以没有出现Using index.

如果使用了key,但没有出现Using index,说明索引并不能覆盖检索和核对的所有字段,需要回访表去获取其它字段内容,这相对于覆盖索引增加了回访表的成本,增加了随机IO的成本

2.4. 通过key_len来确定使用复合索引的几个字段

对于复合索引，比如 INDEX(a,b,c)使用了几个字段，看以观察key_len来确定

```
KEY `idx_appId_createTime_fromPlat` (`appId`,`createTime`,`fromPlat`)
```

首先说明key_len的长度，计算的是字段所占的字节长度：

a) 字段可以为NULL时，需要一个额外字节标记该字段是否为NULL,不为NULL时，不需要这一个额外字节

b) 当字段为varchar类型时，需要额外2个字节标记varchar类型的长度，所以varchar(n)的key_len长度为 3*n + 2 (utf-8)

c) int 4个字节， bigint 8个字节

```
Create Table: CREATE TABLE `operationMenuInfo` (
  `id` int(50) NOT NULL AUTO_INCREMENT,
  `operationMenuName` varchar(200) NOT NULL,
  `createTime` int(50) DEFAULT NULL,
  `startTime` int(50) DEFAULT NULL,
  `endTime` int(50) DEFAULT NULL,
  `appId` int(50) NOT NULL,
  `status` int(50) NOT NULL,
  `fromPlat` varchar(200) DEFAULT NULL,
  `appName` varchar(200) DEFAULT NULL,
  `packageId` int(20) DEFAULT NULL,
  `menuType` smallint(5) NOT NULL DEFAULT '0' COMMENT 'type',
  `entityId` int(11) NOT NULL DEFAULT '0' COMMENT 'entityId',
  `productId` int(11) NOT NULL DEFAULT '0' COMMENT 'pid',
  PRIMARY KEY (`id`),
  KEY `time_appid` (`appId`,`createTime`),
  KEY `idx_startTime` (`startTime`),
  KEY `idx_endTime` (`endTime`),
  KEY `t_eId_pId` (`entityId`,`menuType`,`productId`),
  KEY `idx_appId_createTime_fromPlat` (`appId`,`createTime`,`fromPlat`)
) ENGINE=InnoDB AUTO_INCREMENT=4656258 DEFAULT CHARSET=utf8
```

```
mysql> explain select appId,createTime,fromPlat,status from operationMenuInfo where appId=927 and createTime=1494492062;
+----+-------------+-----------------+------+-----------------------------------------+-------------+---------+-------------+------+-------+
| id | select_type | table           | type | possible_keys                           | key         | key_len | ref         | rows | Extra |
+----+-------------+-----------------+------+-----------------------------------------+-------------+---------+-------------+------+-------+
|  1 | SIMPLE      | operationMenuInfo | ref  | time_appid,idx_appId_createTime_fromPlat | time_appid | 9       | const,const |    1 | NULL  |
+----+-------------+-----------------+------+-----------------------------------------+-------------+---------+-------------+------+-------+
1 row in set (0.00 sec)
```

根据最右前缀，应该用到 两个字段索引

key_len长度： 4 + （4 + 1） = 9

```
mysql> explain select appId,createTime,fromPlat,status from operationMenuInfo where appId=927 and fromPlat='dataman';
+----+-------------+-----------------+------+-----------------------------------------+-------------+---------+-------+--------+-------------+
| id | select_type | table           | type | possible_keys                           | key         | key_len | ref   | rows   | Extra       |
+----+-------------+-----------------+------+-----------------------------------------+-------------+---------+-------+--------+-------------+
|  1 | SIMPLE      | operationMenuInfo | ref  | time_appid,idx_appId_createTime_fromPlat | time_appid | 4       | const | 753128 | Using where |
+----+-------------+-----------------+------+-----------------------------------------+-------------+---------+-------+--------+-------------+
1 row in set (0.00 sec)
```

根据最左前缀，应该用到1个字段的索引

key_len长度： 4

2.5. order by和using filesort

业务SQL经常会有order by,一般来说这需要真实的物理排序才能达到这个效果,这就是我们所说的Using filesort,一般来说它需要检索出所有的符合where条件的数据记录,而后在内存/文件层面进行物理排序,所以一般是一个很耗时的操作,是我们极力想要避免的.

但其实对于MySQL来说,却不一定非得物理排序才能达到order by的效果,也可以通过索引达到order by的效果,却不需要物理排序.

因为索引通过叶节点上的双向链表实现了逻辑有序性,比如说对于where a=? order by b limit 1; 可以直接使用index(a,b)来达到效果,不需要物理排序从索引的根节点,走到叶节点,找到a=?的位置,因为这时b是有序的,只要顺着链表向右走,扫描1个位置,就可以找到想要的1条记录,这样既达到了业务SQL的要求,也避免了物理的排序操作。**这种情况下,执行计划的Extra部分就不会出现Using filesort,因为它只扫描了极少量的索引叶节点就返回了结果,所以一般而言,执行很快,资源消耗很少,是我们想要的效果.**

```
mysql> explain select * from operationMenuInfo where appId=927 order by createTime desc limit 1;
+----+-------------+-------------------+------+-----------------------------------------+-------------+---------+-------+--------+-------------+
| id | select_type | table             | type | possible_keys                           | key         | key_len | ref   | rows   | Extra       |
+----+-------------+-------------------+------+-----------------------------------------+-------------+---------+-------+--------+-------------+
|  1 | SIMPLE      | operationMenuInfo | ref  | time_appid,idx_appId_createTime_fromPlat | time_appid | 4       | const | 783590 | Using where |
+----+-------------+-------------------+------+-----------------------------------------+-------------+---------+-------+--------+-------------+
1 row in set (0.00 sec)

mysql> explain select * from operationMenuInfo where appId=927 order by packageId desc limit 1;
+----+-------------+-------------------+------+-----------------------------------------+-------------+---------+-------+--------+-----------------------------+
| id | select_type | table             | type | possible_keys                           | key         | key_len | ref   | rows   | Extra                       |
+----+-------------+-------------------+------+-----------------------------------------+-------------+---------+-------+--------+-----------------------------+
|  1 | SIMPLE      | operationMenuInfo | ref  | time_appid,idx_appId_createTime_fromPlat | time_appid | 4       | const | 783680 | Using where; Using filesort |
+----+-------------+-------------------+------+-----------------------------------------+-------------+---------+-------+--------+-----------------------------+
1 row in set (0.00 sec)
```

因为存在KEY `time_appid` (`appId`,`createTime`), 第1个SQL可以通过它快速的返回结果,因为没有物理排序,所以执行计划的Extra部分没有出现Using filesort.

而第2个SQL是无法通过任何索引达到上述效果的,必须扫描出所有的符合条件的记录行后物理排序再返回TOP1的记录,因为存在物理排序,所以执行计划的Extra部分出现了Using filesort.

执行时间上,第1个SQL瞬间返回结果,第2个SQL需要0.7秒左右才能返回结果(因为它要检索出符合条件的40W记录,而后还要排序,这2个操作导致了它执行时间偏长).

3.索引设计注意事项

3.1. 关于INNODB表PRIMARY KEY的建议

表设计层面,我们一般建议使用自增ID做PRIMARY KEY,业务主键做UNIQUE KEY,原因如下:

1.如果业务主键做PRIMARY KEY,业务主键的插入顺序比较随机,这样会导致插入时间偏长,而且聚簇索引叶节点分裂严重,导致碎片严重,浪费空间;而自增ID做PRIMARY KEY的情况下,顺序插入,插入快,而且聚簇索引比较紧凑,空间浪费小。

2.一般表设计上除了PRIMARY KEY外,还会有几个索引用来优化读写.而这些非PK索引叶节点中都要存储PRIMARY KEY,以指向数据行,从而关联非索引中的字段内容.这样自增ID（定义为bigint才占用8个字节）和业务主键(通常字符串,多字段,空间占用大)相比,做PRIMARY KEY在索引空间层面的优势也是很明显的(同时也会转换为时间成本层面的优势),表定义中的索引越多,这种优势越明显。

3.2. 什么列上适合建索引,什么列上不适合建索引

这里涉及到一个重要的概念:字段的选择性

select count(1)/count(distinct col) 这个结果越接近数据总行数,那么这个字段的选择性越低; 越接近1,那么这个字段的选择性越高. 简单举例说就是:身份证ID字段的选择性很高,而性别字段的选择性很低.

为什么在低选择性的字段上不适合建立索引？——> 引出另一个问题：**使用索引一定比全表扫描要好吗? 答案是否定的.**

public_status字段分布：

```
mysql> select public_status,count(1) from novel_agg_info group by public_status;
+---------------+----------+
| public_status | count(1) |
+---------------+----------+
|             0 |  3511945 |
|             1 |   367234 |
|             2 |   419062 |
|            12 |       16 |
+---------------+----------+
4 rows in set (1.35 sec)
```

两条SQL

```
mysql> select sql_no_cache count(1) from (select * from novel_agg_info where public_status = 0) tmp;
+----------+
| count(1) |
+----------+
|  3511945 |
+----------+
1 row in set (11.59 sec)

mysql> select sql_no_cache count(1) from (select * from novel_agg_info ignore index(idx_public_status)  where public_status = 0) tmp;
+----------+
| count(1) |
+----------+
|  3511945 |
+----------+
1 row in set (8.46 sec)
```

为什么全表扫描反而快了,使用索引反而慢了呢?

一定程度上是因为回访表的操作,使用索引,但提取了索引字段外的数据,所以需要回访表数据,这里符合条件的数据量特别大,所以导致了大量的回表操作,带来了大量的随机IO; 而全表扫描的话,虽然说表空间比索引空间大,但可以使用多块读特性,一定程度上使用顺序读; 此消彼长,导致全表扫描反而比使用索引还要快了.

3.3. 索引一定是有益的吗?

答案是否定的,因为**索引是有代价的**:

每次的写操作,都要维护索引,相应的调整索引数据,会在一定程度上降低写操作的速度.所以大量的索引必然会降低写性能,索引的创建要从整体考虑,在读写性能之间找到一个好的平衡点,在主要矛盾和次要矛盾之间找到平衡点.

所以说,索引并不是越多越好,无用的索引要删除,冗余的索引(这在后面会提到)要删除,因为它们只有维护上的开销,却没有益处,所以在业务逻辑,SQL,索引结构变更的时候,**要及时删除无用/冗余的索引.**

索引使用不合理的情况下,使用索引也不一定会比全表扫描快,上面也提到了.

总结说,索引不是万能的,要合理的创建索引.

3.4. where条件中不要在索引字段侧进行任何运算(包括隐式运算),否则会导致索引不可用,导致全表扫描

select * from tab where id + 1 = 1000; 会导致全表扫描,应该修改为select * from tab where id = 1000 -1; 才可以高效返回.

select * from tab where from_unixtime(addtime) = '2017-05-11 00:00:00' 会导致index(addtime)不可用

应该调整为select * from tab where addtime = unix_timestamp('2017-05-11 00:00:00') 这样才可以使用index(addtime)

这里的运算也包括隐式的运算,比如说隐式的类型转换..业务上经常有类型不匹配导致隐式的类型转换的情况.这里经常出现的情况是字符串和整型比较.

（即你的字段类型是整型，你传进来一个字符串；你的字段类型是字符串，但你传进来个整型），这种情况下可能会导致索引不可用

为什么说可能呢?

这取决于这种隐式的类型转换发生在了哪侧?是表字段侧,还是业务传入数据侧?

a) 字段整型，传入字符串

两者比较发生在 数据侧，字符串转为整型后再进行比较，所以索引仍然可用

b) 字段字符串，传入整型

两者的比较，隐式类型转化发生在字段侧，会导致索引不可用，从而进行全表扫描

所以，在实际业务中，能够使用bigint/int定义的字段均使用bigint/int定义，这种情况下相对于 长的纯数字字符串的varchar类型来说，更节省空间，而且即使类型不匹配了，也不至于使 索引失效

但隐式的类型转换照样是有性能损耗的,所以还是一致的好

注意：mysql字符串比较默认是不区分大小写的，所以有些业务上为了严格匹配,区分大小写,在SQL中使用了binary,确实达到了区分大小写的目的,但导致索引不可用了(因为在字段侧进行了运算)

表定义中存在合适的索引 `KEY `idx_app_name_status` (`appname`,`status`)`

```
mysql> explain select * from tbl_rtlc_conf where binary appname='LbsPCommon' and status = 1;
+----+-------------+---------------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table         | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+---------------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | tbl_rtlc_conf | ALL  | NULL          | NULL | NULL    | NULL | 9156 | Using where |
+----+-------------+---------------+------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

但因为binary的使用,导致了全表扫描.

mysql的字符串比较默认不区分大小写,是因为它们默认的collation是不区分大小写的

utf8的默认collation为utf8_general_ci,它是不区分大小写的

所以为了严格匹配中即区分大小写又不至于让索引失效，我们可以针对表中具体字段的collation进行调整

`appname` varchar(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL COMMENT 'appname'

3.5. 不要使用%xxx%这种模糊匹配,会导致全表扫描

where name like '%zhao%' **这种前后统配的模糊查询,会导致索引不可用**

如果确实存在这样高频执行的模糊匹配的业务需求,建议走全文检索系统,不要使用MySQL来做这个事情.

where name like 'xxx%' 这种,不前统配,只后统配的,确实是可以使用索引的.

3.6. 关于前缀索引和冗余索引

index(a,b,c) 能同时优化下面几类查询:

where a=? and b=? and c=?

where a=? and b=?

where a=?

也能优化如下的排序查询:

where a=? order by b limit

where a=? and b=? order by c limit

但不能优化 where b=? and c=? 因为索引定义index(a,b,c) 的前缀列a没有出现在where条件中.

更不能优化where c=?

对于where a=? and c=? 查询,它只能使用index(a,b,c)的第1个索引字段a.

所以,如果业务查询为如下2类:

where b=? and c=?

where c=?

那么就应该定义索引为index(c,b),它能同时优化上面2类查询 ,而不应该定义索引index(b,c)的,因为索引index(b,c)优化不了where c=? 因为这个索引的前缀列b没有出现在where条件中.

也不建议创建2个索引: index(b,c) 和index(c) 因为前面提到了索引越少越好,可以用一个index(c,b) 来完成的,就不要创建2个索引来完成.

上面提到了绝大多数情况下,冗余了,可以DROP了,但也存在例外的情况,它们的存在还是必要的:

那就是存在下面的查询:

where a=? order by id [limit] (这里id是表的primary key)

这里index(a) (实际为index(a,id)) 可以优化上面的查询,通过使用这个索引,避免物理排序而达到排序的实际效果.

但index(a,b,c) (实际为index(a,b,c,id)) 和index(a,b) (实际为index(a,b,id)) 却达不到这样的效果.

这种情况下,存在index(a,b,c)的情况下,index(a) 是不冗余的,是需要保留的.

如果不存在这种情况,存在index(a,b,c)的情况下,index(a) ,index(a,b) 都是冗余的,建议drop掉.

但如果where a=? 后返回的数据行已经很少,也就是说对很少的数据进行order by id排序的话,也是可以使用index(a,b)或者index(a,b,c) 来过滤行的,只不过还需要进行物理排序,但代价已经很小了,是否还需要创建一个index(a)需要业务折中考虑了.

3.7. 关于索引定义中的字段顺序

建议where条件中等值匹配的字段放到索引定义的前部,范围匹配的字段(> < between in等为范围匹配)放到索引定义的后面.

因为前缀索引字段使用了范围匹配后,会导致后续的索引字段不能高效的用于优化查询.

3.8. 关于单列索引和复合索引

有时候会看到业务SQL是where a=? and b=? and c=?

但3个列上分别创建了一个单列索引:

index(a)　index(b)　index(c)

这种创建是否合理呢?

前面提到**高选择性字段上适合创建索引,低选择性字段上不适合创建单列索引(但可以考虑作为复合索引定义的一部分)**

如果a字段上的选择性足够高,b,c的选择性低,完全可以只创建索引index(a) , 这种情况下,当然也可以只创建index(a,b) 或者只创建index(a,b,c). (不要创建index(b), index(c) 这2个低选择性字段上的单列索引了).

如果实际情况是a,b,c单独的选择性一般,都不是很高,但3个组合到一起的选择性很高的话,那就建议创建index(a,b,c)的组合索引,不要3个字段上都创建一个单列索引.

为什么呢？ mysql确实可以使用index merge来使用多个索引,但很多时候是否比得上复合索引效率高呢?

简化一下: where a=? and b=?

a=? 返回1W行记录, b=? 返回1W行记录, where a=? and b=? 返回100行记录.

如果是两个单列索引: index(a) index(b) 的情况下,index_merge会是一个什么样的执行计划呢?

针对a=? 通过使用index(a) 返回1W行记录,带PRMIARY KEY

针对b=? 通过使用index(b) 返回1W行记录,带PRMIARY KEY

然后对primary key 取交集,不管是排序后取交集也好,还是通过嵌套循环,关联的方式取交集也好.都会是一个耗时耗费资源的操作.

综合来说,扫描各自的索引返回1W行记录,而后对这2W行记录取交集,肯定是一个耗时耗费资源的操作了.

但如果存在复合索引index(a,b) 通过索引的扫描定位,可以快速的返回这100行记录的.

所以针对这种情况,建议创建复合索引,不要创建多个单列索引.

4. 1个优化案例

```
mysql> show create table lc_day_channel_version\G
*************************** 1. row ***************************
       Table: lc_day_channel_version
Create Table: CREATE TABLE `lc_day_channel_version` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT COMMENT '',
  `prodline` varchar(50) NOT NULL DEFAULT '' COMMENT '',
  `os` tinyint(3) unsigned NOT NULL DEFAULT '0' COMMENT '1:Android_Phone 2:
Android_Pad 3:IPhone',
  `original_type` tinyint(3) unsigned NOT NULL DEFAULT '0' COMMENT '1 3',
  `dtime` int(10) unsigned NOT NULL DEFAULT '0' COMMENT 'date time,
yyyymmdd',
  `version_name` varchar(50) NOT NULL DEFAULT '' COMMENT '',
  `channel` varchar(50) NOT NULL DEFAULT '' COMMENT '',
  `request_pv` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '',
  `request_uv` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '',
  `response_pv` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '',
  `response_uv` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '',
  `download_pv` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '',
  `download_uv` bigint(20) unsigned NOT NULL DEFAULT '0' COMMENT '',
```

```
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQUE_poouvc` (`prodline`,`os`,`original_type`,`dtime`,
`version_name`,`channel`),
  KEY `INDEX_d` (`dtime`)
) ENGINE=InnoDB AUTO_INCREMENT=135293125 DEFAULT CHARSET=utf8 COMMENT=''
1 row in set (0.00 sec)

mysql> explain select version_name,sum(request_pv) as request_pv from
lc_day_channel_version where dtime>=20170504 and dtime<=20170510 group by
version_name order by request_pv desc;
+----+-------------+-----------------------+-------
+----------------------+---------+---------+------+---------
+-----------------------------------------------------------+
| id | select_type | table                 | type  |
possible_keys         | key     | key_len | ref  | rows    |
Extra                                                       |
+----+-------------+-----------------------+-------
+----------------------+---------+---------+------+---------
+-----------------------------------------------------------+
|  1 | SIMPLE      | lc_day_channel_version | range | UNIQUE_poouvc,
INDEX_d | INDEX_d | 4       | NULL | 2779470 | Using index condition;
Using temporary; Using filesort |
+----+-------------+-----------------------+-------
+----------------------+---------+---------+------+---------
+-----------------------------------------------------------+
1 row in set (0.00 sec)


SQL,4.7s.

mysql> select count(1) from lc_day_channel_version where dtime>=20170504
and dtime<=20170510;
+----------+
| count(1) |
+----------+
|  1462991 |
+----------+
1 row in set (0.58 sec)
146W,index(dtime),version_name,request_pv,146WIO + IO,
146W group by version_name order by request_pv desc,.
,:alter table lc_day_channel_version add key
idx_dtime_version_name_request_pv(dtime,version_name,request_pv);
:
mysql> explain select sql_no_cache version_name,sum(request_pv) as
request_pv from lc_day_channel_version where dtime>=20170504 and
dtime<=20170510 group by version_name order by request_pv desc;
+----+-------------+-----------------------+-------
+-----------------------------------------------------------
+----------------------------------+---------+------+---------
+-----------------------------------------------------------+
| id | select_type | table                 | type  |
possible_keys                                    |
key                              | key_len | ref  | rows    |
```

```
Extra                                                          |
+----+-------------+-----------------------+-------
+-------------------------------------------------------------
+-----------------------------------+---------+------+---------
+-------------------------------------------------------------+
|  1 | SIMPLE      | lc_day_channel_version | range | UNIQUE_poouvc,
INDEX_d,idx_dtime_version_name_request_pv |
idx_dtime_version_name_request_pv | 4       | NULL | 2681154 | Using
where; Using index; Using temporary; Using filesort |
+----+-------------+-----------------------+-------
+-------------------------------------------------------------
+-----------------------------------+---------+------+---------
+-------------------------------------------------------------+
1 row in set (0.00 sec)
Using index ,,2.35s.



index(dtime,version_name,request_pv)
dtime,.
```

多列索引的好处：

使用复合索引进行查询

1.最左前缀原则

2.mysql引擎在查询时为了更好的利用索引，会动态的调整字段顺序来利用索引

比如：KEY(firstname,lastname,age) 一个索引相当于

KEY(firstname) KEY(firstname,lastname) KEY(firstname,lastname,age) 三个索引

where => firstname ,lastname,age 可以

firstname,lastname 可以

firstname 可以

lastname, firstname 时？ 可以

firstname,age 时？ 不可以

age ? 不可以

explain这个命令来查看一个这些SQL语句的执行计划，查看该SQL语句有没有使用上了索引，有没有做全表扫描，这都可以通过explain命令来查看。

(2) 为了建立有效索引，提高索引覆盖，把所有列均建上索引?

首先索引不是越多越好，

索引是需要占用物理空间的，而且创建和维护索引都需要时间，在对数据进行增删改时都需要维护索引

过多的索引会维护过多的数据，会严重影响增删改的性能

```
Select age FROM people Where firstname='Mike' AND lastname='Sullivan'; age
```

(3) 从innodb的索引结构分析，为什么索引的key长度不能太长?

key太长会导致一个页当中能够存放的key的数目变少，间接导致索引树的页数目变多，索引层次增加，从而影响整体查询变更的效率。

(4) 索引构建原则：

索引虽好，但也不是无限制的使用，最好符合一下几个原则

1） 最左前缀匹配原则，组合索引非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2）较频繁作为查询条件的字段才去创建索引

3）更新频繁字段不适合创建索引

4）若是不能有效区分数据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)

5）尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

7）对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。

8）对于定义为text、image和bit的数据类型的列不要建立索引。

参考资料：

https://zhuanlan.zhihu.com/p/350863953

https://zhuanlan.zhihu.com/p/277567026

MYSQL索引设计的注意事项#%E4%BA%94%E3%80%81%E5%87%A0%E4%B8%AA%E4%BC%98%E5%8C%96%E6%A1%88%E4%BE%8B