# 索引与执行计划

## 索引入门

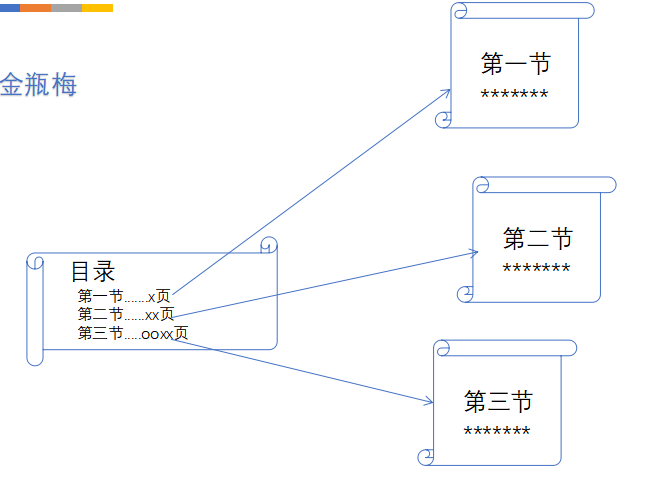
### 索引是什么

#### 生活中的索引

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。

可以得到索引的本质：**索引是数据结构**。

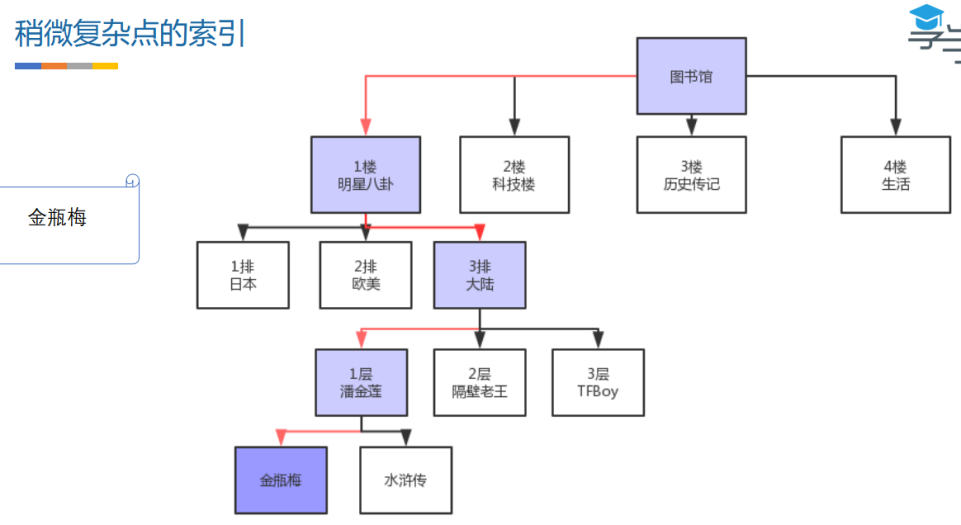
上面的理解比较抽象，举一个例子，平时看任何一本书，首先看到的都是目录，通过目录去查询书籍里面的内容会非常的迅速。



上图就是一本金瓶梅的书，书籍的目录是按顺序放置的，有第一节，第二节它本身就是一种顺序存放的数据结构，是一种顺序结构。

另外通过目录（索引），可以快速查询到目录里面的内容，它能高效获取数据，通过这个简单的案例可以理解所以就是**高效获取数据的数据结构**

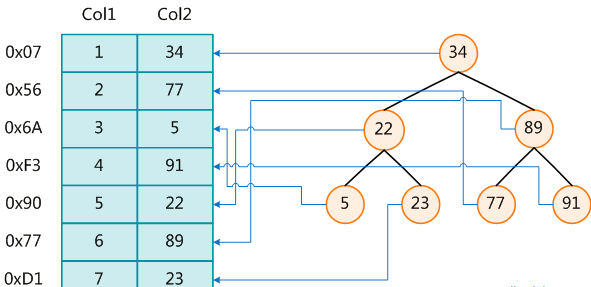
再来看一个发杂的情况，



我们要去图书馆找一本书，这图书馆的书肯定不是线性存放的，它对不同的书籍内容进行了分类存放，整索引由于一个个节点组成，根节点有中间节点，中间节点下面又由子节点，最后一层是叶子节点，

可见，整个索引结构是一棵倒挂着的树，其实它就是一种数据结构，这种数据结构比前面讲到的线性目录更好的增加了查询的速度。

#### MySql中的索引



MySql中的索引其实也是这么一回事，我们可以在数据库中建立一系列的索引，比如创建主键的时候默认会创建主键索引，上图是一种BTREE的索引。每一个节点都是主键的ID

当我们通过ID来查询内容的时候，首先去查索引库，在到索引库后能快速的定位索引的具体位置。

### 索引得分类

**普通索引：**即一个索引只包含单个列，一个表可以有多个单列索引

**唯一索引：**索引列的值必须唯一，但允许有空值

**复合索引：**即一个索引包含多个列

**聚簇索引(聚集索引)**：并不是一种单独的索引类型，而是一种数据存储方式。具体细节取决于不同的实现，InnoDB的聚簇索引其实就是在同一个结构中保存了B-Tree索引(技术上来说是B+Tree)和数据行。

**非聚簇索引：**不是聚簇索引，就是非聚簇索引

### 基础语法

**查看索引**

SHOW INDEX FROM table\_name\G

**创建索引**

CREATE [UNIQUE ] INDEX indexName ON mytable(columnname(length));

ALTER TABLE 表名 ADD [UNIQUE ] INDEX [indexName] ON (columnname(length))

**删除索引**

DROP INDEX [indexName] ON mytable;

## 执行计划

### 什么是执行计划

使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是

如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈

### 执行计划的作用

* 表的读取顺序
* 数据读取操作的操作类型
* 哪些索引可以使用
* 哪些索引被实际使用
* 表之间的引用
* 每张表有多少行被优化器查询

以上的这些作用会在执行计划详解里面介绍到，在这里不做解释。

### 执行计划的语法

执行计划的语法其实非常简单： 在SQL查询的前面加上EXPLAIN关键字就行。

比如：EXPLAIN select \* from table1

重点的就是EXPLAIN后面你要分析的SQL语句

### 执行计划详解

通过EXPLAIN关键分析的结果由以下列组成，接下来挨个分析每一个列



#### ID列

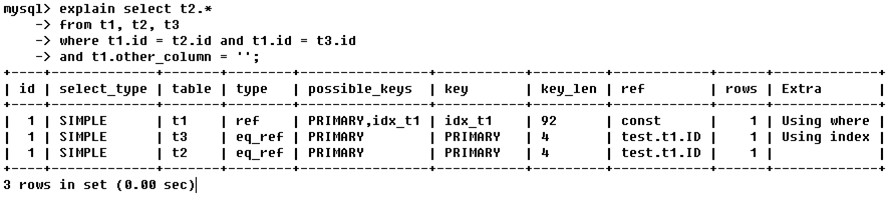
ID列：描述select查询的序列号,包含一组数字，表示查询中执行select子句或操作表的顺序

根据ID的数值结果可以分成一下三种情况

* id相同：执行顺序由上至下
* id不同：如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行
* id相同不同：同时存在

分别举例来看

Id相同



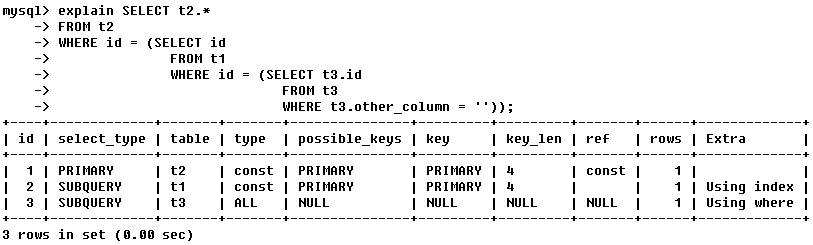
如上图所示，ID列的值全为1，代表执行的允许从t1开始加载，依次为t3与t2

EXPLAIN

select t2.\* from t1,t2,t3 where t1.id = t2.id and t1.id = t3.id

and t1.other\_column = '';

Id不同



如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行

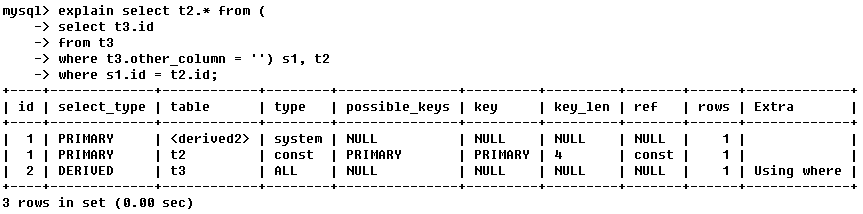
EXPLAIN

select t2.\* from t2 where id = (

select id from t1 where id = (select t3.id from t3 where t3.other\_column='')

);

Id相同又不同



id如果相同，可以认为是一组，从上往下顺序执行；

在所有组中，id值越大，优先级越高，越先执行

EXPLAIN

select t2.\* from (

select t3.id

from t3 where t3.other\_column = ''

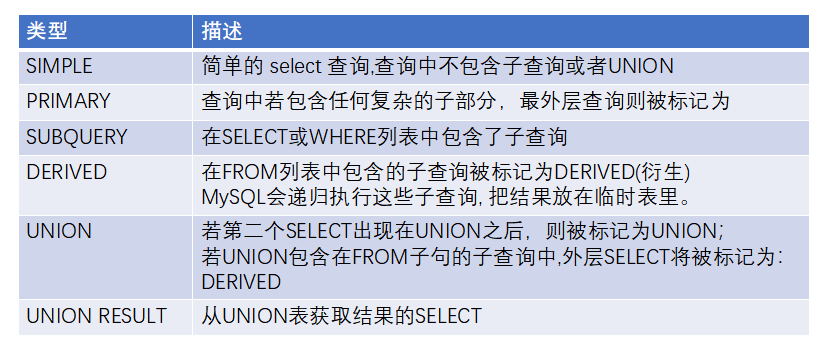
) s1 ,t2 where s1.id = t2.id

#### select\_type列

Select\_type:查询的类型，

要是用于区别:普通查询、联合查询、子查询等的复杂查询

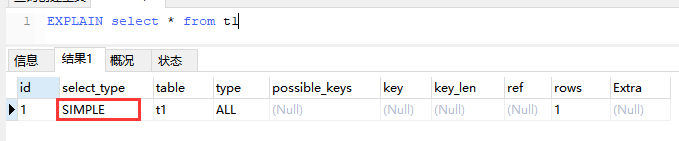
类型如下



SIMPLE

EXPLAIN select \* from t1

简单的 select 查询,查询中不包含子查询或者UNION



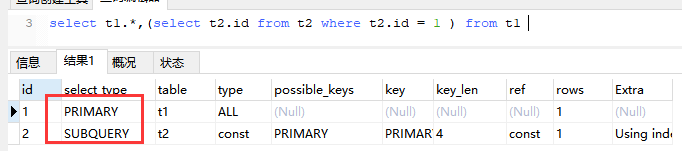
PRIMARY与SUBQUERY

PRIMARY：查询中若包含任何复杂的子部分，最外层查询则被标记为

SUBQUERY：在SELECT或WHERE列表中包含了子查询

EXPLAIN

select t1.\*,(select t2.id from t2 where t2.id = 1 ) from t1

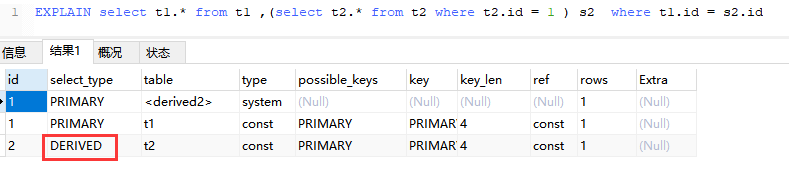


DERIVED

在FROM列表中包含的子查询被标记为DERIVED(衍生)

MySQL会递归执行这些子查询, 把结果放在临时表里。

select t1.\* from t1 ,(select t2.\* from t2 where t2.id = 1 ) s2 where t1.id = s2.id



UNION RESULT 与UNION

UNION：若第二个SELECT出现在UNION之后，则被标记为UNION；

UNION RESULT：从UNION表获取结果的SELECT

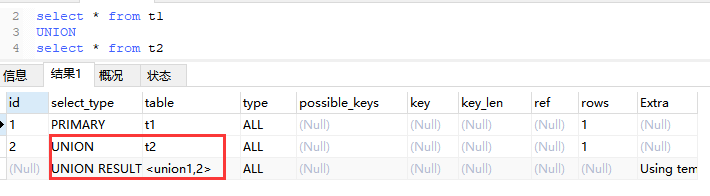
#UNION RESULT ,UNION

EXPLAIN

select \* from t1

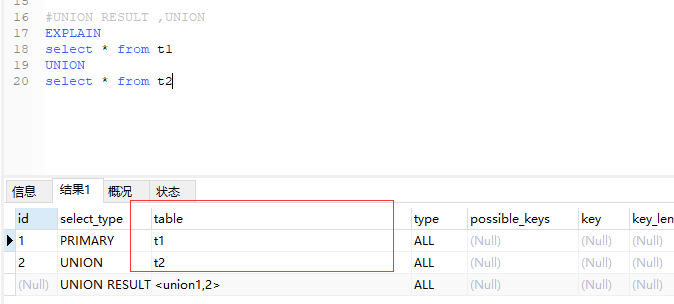
UNION

select \* from t2



#### table列

显示这一行的数据是关于哪张表的



#### Type列

type显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

需要记忆的

system>const>eq\_ref>ref>range>index>ALL

一般来说，得保证查询至少达到range级别，最好能达到ref。

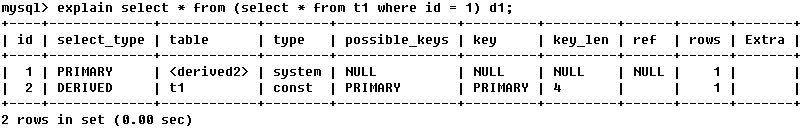
System与const

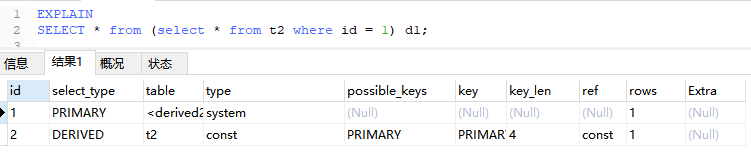
System：表只有一行记录（等于系统表），这是const类型的特列，平时不会出现，这个也可以忽略不计

Const：表示通过索引一次就找到了

const用于比较primary key或者unique索引。因为只匹配一行数据，所以很快

如将主键置于where列表中，MySQL就能将该查询转换为一个常量



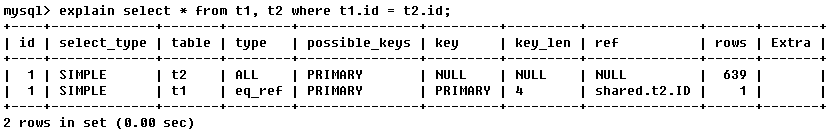


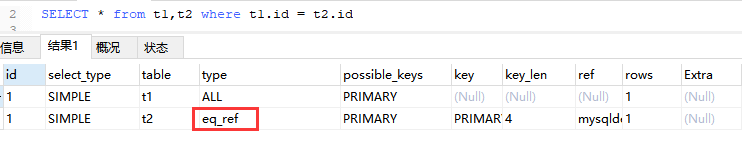
EXPLAIN

SELECT \* from (select \* from t2 where id = 1) d1;

eq\_ref

唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描





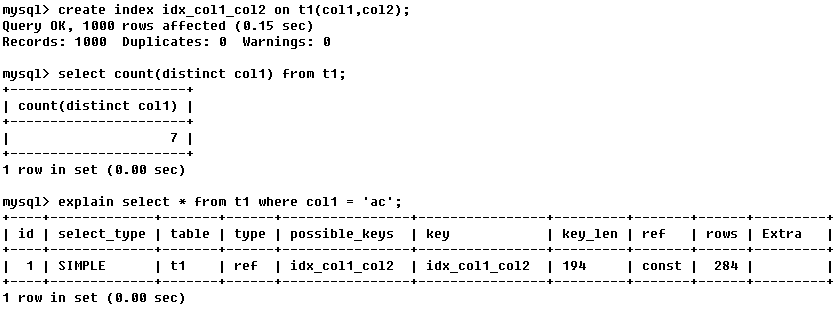
**EXPLAIN**

**SELECT \* from t1,t2 where t1.id = t2.id**

Ref

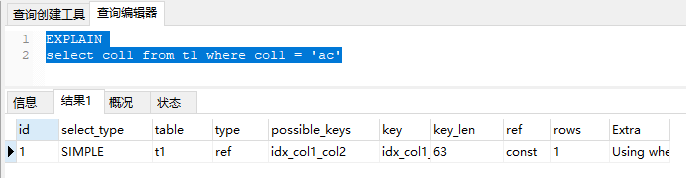
非唯一性索引扫描，返回匹配某个单独值的所有行.

本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而，它可能会找到多个符合条件的行，所以他应该属于查找和扫描的混合体



EXPLAIN

select count(DISTINCT col1) from t1 where col1 = 'ac'



或者

EXPLAIN

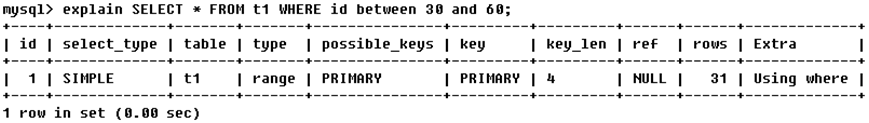
select col1 from t1 where col1 = 'ac'

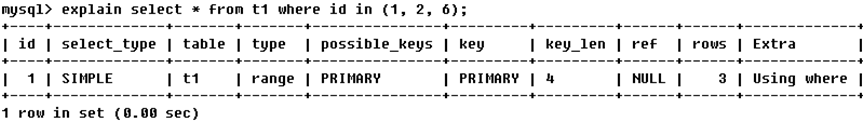
Range

只检索给定范围的行,使用一个索引来选择行。key 列显示使用了哪个索引

一般就是在你的where语句中出现了between、<、>、in等的查询

这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一点，不用扫描全部索引。





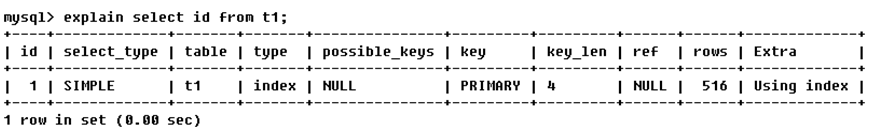
EXPLAIN select \* from t1 where id BETWEEN 30 and 60

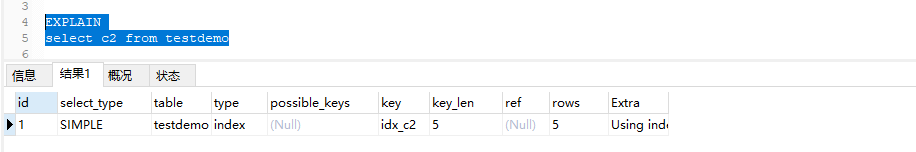
EXPLAIN select \* from t1 where id in(1,2)

Index

当查询的结果全为索引列的时候，虽然也是全部扫描，但是只查询的索引库，而没有去查询

数据。



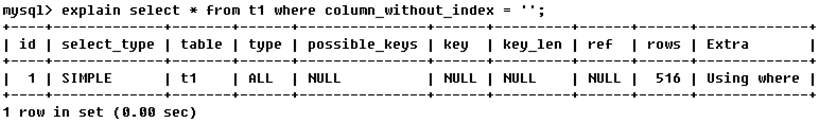


EXPLAIN

select c2 from testdemo

All

Full Table Scan，将遍历全表以找到匹配的行



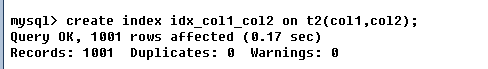
#### possible\_keys 与Key

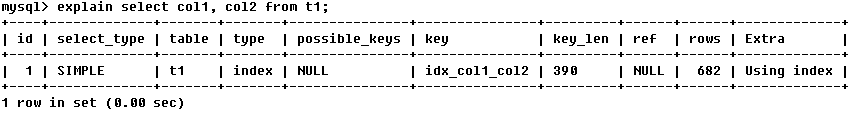
possible\_keys:可能使用的key

Key:实际使用的索引。如果为NULL，则没有使用索引

查询中若使用了**覆盖索引**，则该索引和查询的select字段重叠

这里的覆盖索引非常重要，后面会单独的来讲

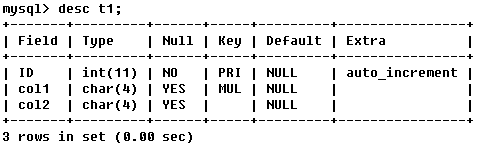


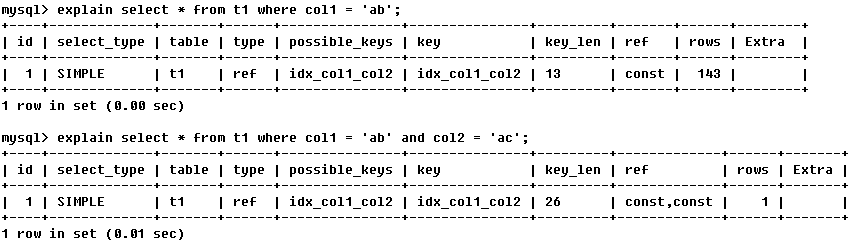


EXPLAIN select col1,col2 from t1

其中key和possible\_keys都可以出现null的情况（结婚邀请朋友的例子）

#### key\_len





desc

select \* from ta where col1 ='ab';

desc

select \* from ta where col1 ='ab' and col2 = 'ac'

Key\_len表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精确性的情况下，长度越短越好

key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的



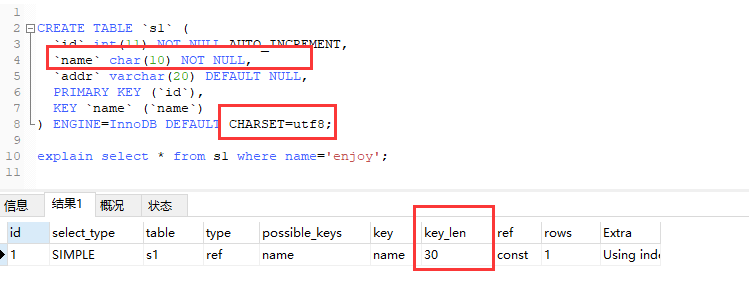
* key\_len表示索引使用的字节数，
* 根据这个值，就可以判断索引使用情况，特别是在组合索引的时候，判断所有的索引字段是否都被查询用到。
* char和varchar跟字符编码也有密切的联系,
* **latin1占用1个字节，gbk占用2个字节，utf8占用3个字节。（不同字符编码占用的存储空间不同）**

字符类型



以上这个表列出了所有字符类型，但真正建所有的类型常用情况只是CHAR、VARCHAR

字符类型-索引字段为char类型+不可为Null时



CREATE TABLE `s1` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` char(10) NOT NULL,

`addr` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `name` (`name`)

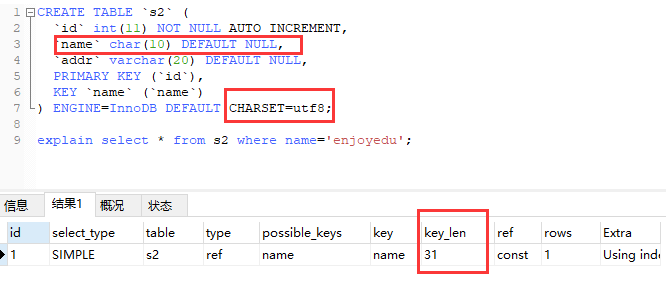
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

explain select \* from s1 where name='enjoy';

name这一列为char(10),字符集为utf-8占用3个字节

Keylen=10\*3

字符类型-索引字段为char类型+允许为Null时



CREATE TABLE `s2` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` char(10) DEFAULT NULL,

`addr` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `name` (`name`)

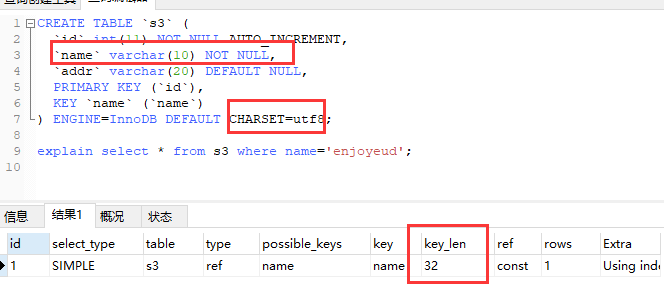
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

explain select \* from s2 where name='enjoyedu';

name这一列为char(10),字符集为utf-8占用3个字节,外加需要存入一个null值

Keylen=10\*3+1(null) 结果为31

索引字段为varchar类型+不可为Null时



CREATE TABLE `s3` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(10) NOT NULL,

`addr` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id`),

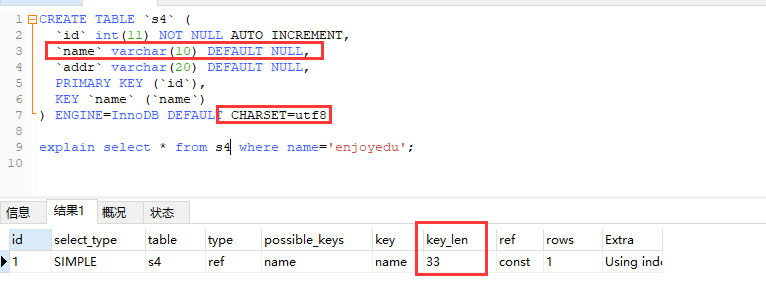
KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

explain select \* from s3 where name='enjoyeud';

Keylen=varchar(n)变长字段+不允许Null=n\*(utf8=3,gbk=2,latin1=1)+2

索引字段为varchar类型+允许为Null时



CREATE TABLE `s3` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(10) NOT NULL,

`addr` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `name` (`name`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

explain select \* from s3 where name='enjoyeud';

Keylen=varchar(n)变长字段+允许Null=n\*(utf8=3,gbk=2,latin1=1)+1(NULL)+2

数值类型



CREATE TABLE `numberKeyLen ` (

`c0` int(255) NOT NULL ,

`c1` tinyint(255) NULL DEFAULT NULL ,

`c2` smallint(255) NULL DEFAULT NULL ,

`c3` mediumint(255) NULL DEFAULT NULL ,

`c4` int(255) NULL DEFAULT NULL ,

`c5` bigint(255) NULL DEFAULT NULL ,

`c6` float(255,0) NULL DEFAULT NULL ,

`c7` double(255,0) NULL DEFAULT NULL ,

PRIMARY KEY (`c0`),

INDEX `index\_tinyint` (`c1`) USING BTREE ,

INDEX `index\_smallint` (`c2`) USING BTREE ,

INDEX `index\_mediumint` (`c3`) USING BTREE ,

INDEX `index\_int` (`c4`) USING BTREE ,

INDEX `index\_bigint` (`c5`) USING BTREE ,

INDEX `index\_float` (`c6`) USING BTREE ,

INDEX `index\_double` (`c7`) USING BTREE

)

ENGINE=InnoDB

DEFAULT CHARACTER SET=utf8 COLLATE=utf8\_general\_ci

ROW\_FORMAT=COMPACT

;

EXPLAIN

select \* from numberKeyLen where c1=1

EXPLAIN

select \* from numberKeyLen where c2=1

EXPLAIN

select \* from numberKeyLen where c3=1

EXPLAIN

select \* from numberKeyLen where c4=1

EXPLAIN

select \* from numberKeyLen where c5=1

EXPLAIN

select \* from numberKeyLen where c6=1

EXPLAIN

select \* from numberKeyLen where c7=1

日期和时间



datetime类型在5.6中字段长度是5个字节

datetime类型在5.5中字段长度是8个字节

CREATE TABLE `datatimekeylen ` (

`c1` date NULL DEFAULT NULL ,

`c2` time NULL DEFAULT NULL ,

`c3` year NULL DEFAULT NULL ,

`c4` datetime NULL DEFAULT NULL ,

`c5` timestamp NULL DEFAULT NULL ,

INDEX `index\_date` (`c1`) USING BTREE ,

INDEX `index\_time` (`c2`) USING BTREE ,

INDEX `index\_year` (`c3`) USING BTREE ,

INDEX `index\_datetime` (`c4`) USING BTREE ,

INDEX `index\_timestamp` (`c5`) USING BTREE

)

ENGINE=InnoDB

DEFAULT CHARACTER SET=utf8 COLLATE=utf8\_general\_ci

ROW\_FORMAT=COMPACT

;

EXPLAIN

SELECT \* from datatimekeylen where c1 = 1

EXPLAIN

SELECT \* from datatimekeylen where c2 = 1

EXPLAIN

SELECT \* from datatimekeylen where c3 = 1

EXPLAIN

SELECT \* from datatimekeylen where c4 = 1

EXPLAIN

SELECT \* from datatimekeylen where c5 = 1

总结

字符类型

变长字段需要额外的2个字节（VARCHAR值保存时只保存需要的字符数，另加一个字节来记录长度(如果列声明的长度超过255，则使用两个字节)，所以VARCAHR索引长度计算时候要加2），固定长度字段不需要额外的字节。

而NULL都需要1个字节的额外空间,所以索引字段最好不要为NULL，因为NULL让统计更加复杂并且需要额外的存储空间。

复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之和，这也可以用来判定复合索引是否部分使用，还是全部使用。

整数/浮点数/时间类型的索引长度

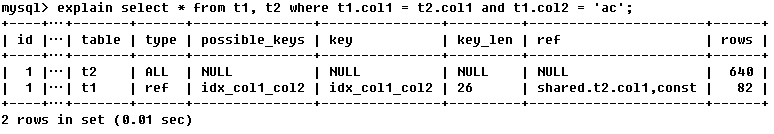
NOT NULL=字段本身的字段长度

NULL=字段本身的字段长度+1(因为需要有是否为空的标记，这个标记需要占用1个字节)

datetime类型在5.6中字段长度是5个字节，datetime类型在5.5中字段长度是8个字节

#### Ref

显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列上的值



EXPLAIN

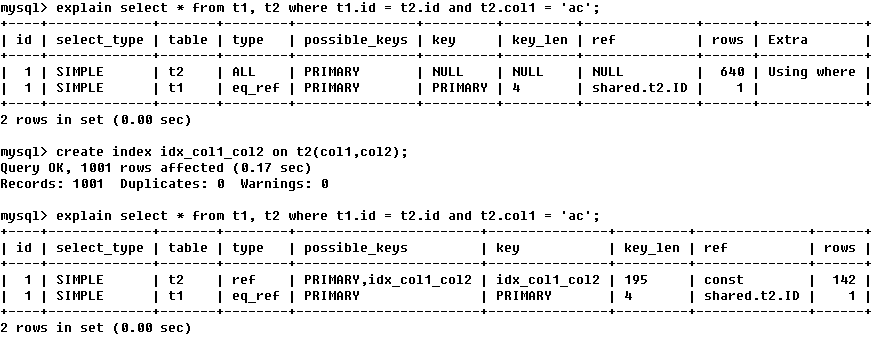
select \* from s1 ,s2 where s1.id = s2.id and s1.name = 'enjoy'

由key\_len可知t1表的idx\_col1\_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac'

其中 【shared.t2.col1】 为 【数据库.表.列】

#### Rows

根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数



#### Extra

包含不适合在其他列中显示但十分重要的额外信息。

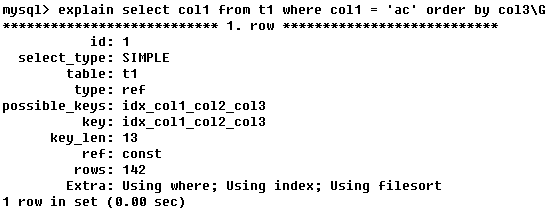


Using filesort

说明mysql会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。

MySQL中无法利用索引完成的排序操作称为“文件排序”

当发现有Using filesort 后，实际上就是发现了可以优化的地方



上图其实是一种索引失效的情况，后面会讲，可以看出查询中用到了个联合索引，索引分别为col1,col2,col3



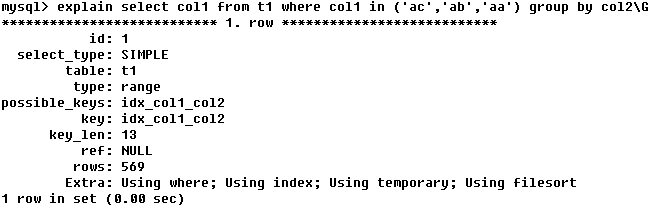
当我排序新增了个col2，发现using filesort 就没有了。

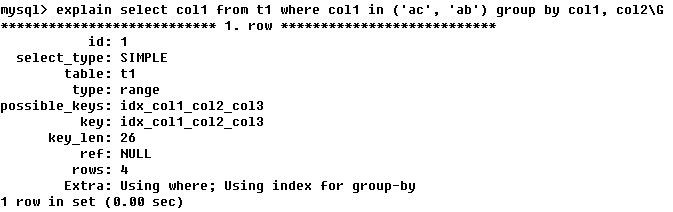
EXPLAIN select col1 from t1 where col1='ac' order by col3

EXPLAIN select col1 from t1 where col1='ac' order by col2,col3

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。





尤其发现在执行计划里面有using filesort而且还有Using temporary的时候，特别需要注意

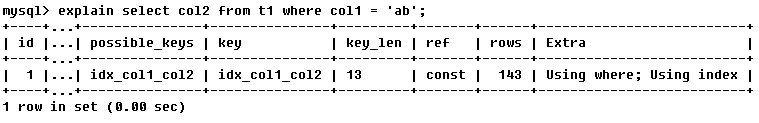
EXPLAIN select col1 from t1 where col1 in('ac','ab','aa') GROUP BY col2

EXPLAIN select col1 from t1 where col1 in('ac','ab','aa') GROUP BY col1,col2

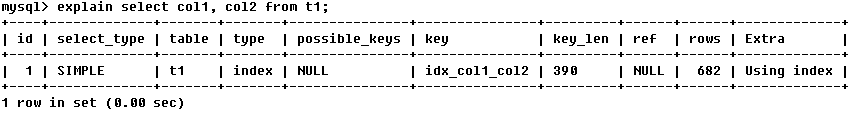
Using index

表示相应的select操作中使用了覆盖索引(Covering Index)，避免访问了表的数据行，效率不错！

**如果同时出现using where，表明索引被用来执行索引键值的查找;**



**如果没有同时出现using where，表明索引用来读取数据而非执行查找动作**



EXPLAIN select col2 from t1 where col1 = 'ab'

EXPLAIN select col2 from t1

覆盖索引：

覆盖索引（Covering Index）,一说为索引覆盖。

理解方式一:就是select的数据列只用从索引中就能够取得，不必读取数据行，MySQL可以利用索引返回select列表中的字段，而不必根据索引再次读取数据文件,换句话说查询列要被所建的索引覆盖。

理解方式二:索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据;当能通过读取索引就可以得到想要的数据，那就不需要读取行了。一个索引包含了(或覆盖了)满足查询结果的数据就叫做覆盖索引

注意：

如果要使用覆盖索引，一定要注意select列表中只取出需要的列，不可select \*，

因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降。

所以，千万不能为了查询而在所有列上都建立索引，会严重影响修改维护的性能。

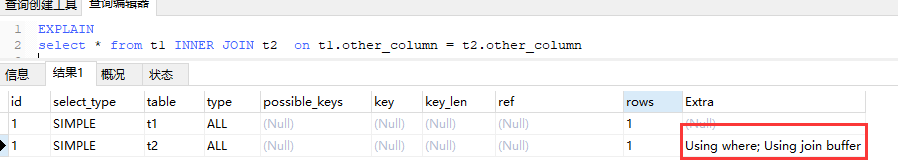
Using where 与 using join buffer

Using where

表明使用了where过滤

using join buffer

使用了连接缓存：

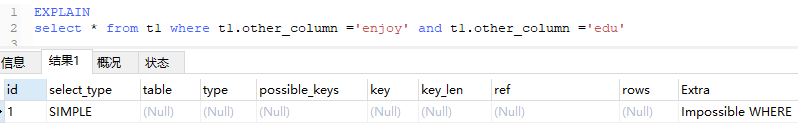


EXPLAIN

select \* from t1 JOIN t2 on t1.other\_column = t2.other\_column

impossible where

where子句的值总是false，不能用来获取任何元组



EXPLAIN

select \* from t1 where 1=2

EXPLAIN

select \* from t1 where t1.other\_column ='enjoy' and t1.other\_column = 'edu'