MySQL 索引及查询优化

# MySQL索引的原理

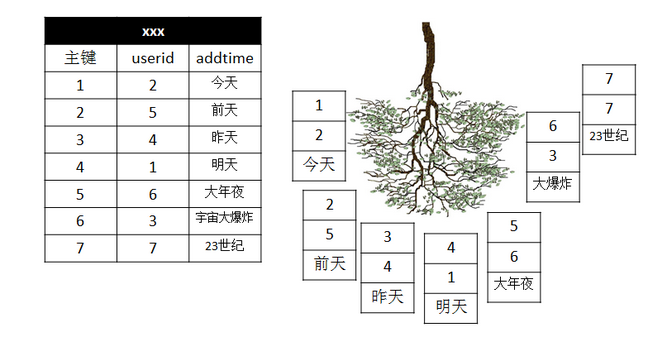
## 索引目的

索引的目的在于提高查询效率，可以类比字典，如果要查“mysql”这个单词，我们肯定需要定位到m字母，然后从下往下找到y字母，再找到剩下的sql。如果没有索引，那么你可能需要把所有单词看一遍才能找到你想要的，如果我想找到m开头的单词呢？或者ze开头的单词呢？

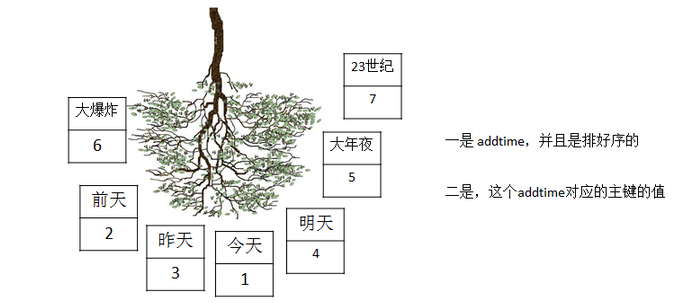
## 索引原理

除了词典，生活中随处可见索引的例子，如火车站的车次表、图书的目录等。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方式来锁定数据。

每个数据表都有一个主键（如果没有主键，数据库会将该表中的唯一索引当作主键使用？），MySQL会以主键的方式构造一棵树，叶子节点存放该主键对应的整行数据。



自己建立的索引，一般叫做辅助索引，辅助索引的树，也自己节点存放了两个东西，一个是索引自身的值，另外一个是索引对应主键的值。



如果索引是联合索引，比如UserID和AddTime索引的方式，索引叶子节点会存储UserID和AddTime之间的配对+主键的配对数据。

## 索引的类型

### B-树索引

B-树索引在生产环境更为广泛，这里我只针对B-树索引进行讨论

B-树索引是一个复杂的内容，可以参见B-tree。

### Hash索引

哈希索引（Hash Index）建立在哈希表的基础上，它只对使用了索引中的每一列的精确查找有用。对于每一行，存储引擎计算出了被索引的哈希码（Hash Code），它是一个较小的值，并且有可能和其他行的哈希码不同。它把哈希码保存在索引中，并且保存了一个指向哈希表中的每一行的指针。在mysql中，只有memory存储引擎支持显式的哈希索引。

* Hash 索引仅仅能满足”=”,”IN”和”<=>”查询，不能使用范围查询。
* Hash 索引无法被用来避免数据的排序操作。
* Hash 索引不能利用部分索引键查询。
* Hash 索引在任何时候都不能避免表扫描。
* Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高

### 空间索引（R-树）索引

主要用于GIS中空间数据的存储，但是MySQL的空间索引支持并不好，现在多使用PostgreSQL。

### 全文索引(Full-text)索引

文本字段上的普通索引只能加快对出现在字段内容最前面的字符串(也就是字段内容开头的字符)进行检索操作。如果字段里存放的是由几个、甚至是多个单词构成 的较大段文字，普通索引就没什么作用了。这种检索往往以LIKE %word%的形式出现，这对MySQL来说很复杂，如果需要处理的数据量很大，响应时间就会很长。

这类场合正是全文索引(full-text index)可以大显身手的地方。在生成这种类型的索引时，MySQL将把在文本中出现的所有单词创建为一份清单，查询操作将根据这份清单去检索有关的数 据记录。全文索引即可以随数据表一同创建，也可以等日后有必要时再使用下面这条命令添加：

ALTER TABLE tablename ADD FULLTEXT(column1, column2)

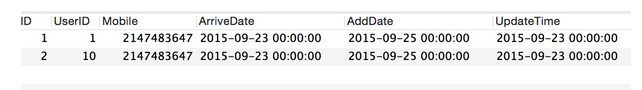
# 索引的合理使用

|  |
| --- |
|  |
| **字段名** | **数据类型** | **NULL** | **INDEX** |
| ID | int(10) | NOT NULL | pk |
| UserID | int(10) | NOT NULL |  |
| Mobile | varchar(15) | NOT NULL |  |
| ArriveDate | DateTime | NOT NULL |  |
| AddDate | DateTime | NOT NULL |  |
| UpdateTime | timetamp | NOT NULL |  |

下面我们的分析都会这个数据表为例。

## 索引在查询中的使用

### 不使用索引



现在刚建立的表上没有任何索引，但是我们想通过UserID找出ArriveDate这个数据，SQL将会写成如下方式：

select ArriveDate from TestSQL where UserID = 10;

这个SQL没有走索引，数据库就会根据主键（ID）扫描全表，每拿到一条数据库记录就与where条件比对，如果符合条件则将这条记录返回，重复直到全表扫描完毕。在大数据量的情况下，不使用索引进行查询几乎是不可行的。

### 使用UserID作为索引

这时候看到where条件是使用了UserID索引的。这时候数据库引擎会根据UserID到索引上找到ID，然后根据ID去查询对应记录，从而取出ArriveDate数据。

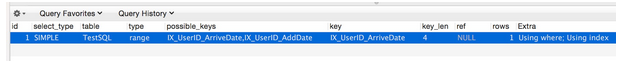


现在我们将UserID的索引更换为UserID,ArriveDate的联合索引,现在再来查询：



可以发现这个Extra里面也是使用了索引的，这就意味这个SQL是完全走了索引，数据库引擎根据UserID找到对应的索引， 因为Select的字段是索引的一部分，所以找到索引之后不需要再读取表记录了。

当一个查询语句中使用设计到多个索引时，MySQL数据库引擎会计算不同索引涉及到的行数大小，选取行数最小的索引作为实际执行时使用的索引，如：



注意：一次查询同一张表，MySQL每次只会使用一个索引。

## 索引在范围查询的使用

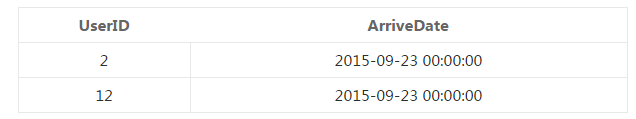
范围查询主要是指查询字段值在某个范围内的记录，表现在where条件中为>,<,between等关键字。如，我们使用如下SQL进行查询：

select UserID,ArriveDate from TestSQL where UserID >0 and UserID <100 and ArriveDate = '2015-09-23 00:00:00';

作为对比，我们使用另一种SQL查询相同记录：

select UserID,ArriveDate from TestSQL where UserID in(2,12) and ArriveDate = '2015-09-23 00:00:00';

联调SQL都会查询出相同的记录：



第一个SQL（使用范围查询）的explain结果为：



第二个SQL（未使用范围查询）的explain结果为：



对比可见，两种sql的索引长度是不一样的。在范围查询中，索引的使用是遵循最左（leftmost）原则，例如这个表的使用的索引IX\_UserID\_ArriveDate，但是因为UserID使用了范围查询(Range query)，就不再使用ArrvieDate的索引了。

## 排序使用索引

在排序中以下情况无法使用索引：

* Order by 的字段并不是索引
* 使用了两种排序方向，但是索引都是使用升序排列的

在索引的原理里我们讲到过：如果索引是联合索引，比如UserID和AddTime索引的方式，索引叶子节点会存储UserID和AddTime之间的配对+主键的配对数据。这种情况下，索引会以UserID进行排序，当UserID相同时再以AddTime进行排序（默认为升序），以我们现有的数据库为例：

select UserID,ArriveDate from TestSQL where UserID in(2,3,4) order by UserID asc,ArriveDate desc;

这种情况将只使用UserID 作为索引，而ArriveDate 将不再作为索引。

不符合最左（Left most）的条件：这种情况将不再使用UserID\_ArriveDate索引

select \* from TestSQL where ArriveDate = '2015-09-23 00:00:00' order by UserID desc

联合查询的情况下，如果第一个筛选条件是范围查询，MySQL不再使用剩下的索引：这种情况下，因为UserID 为范围查询，所以就不会再使用ArriveDate 索引了。

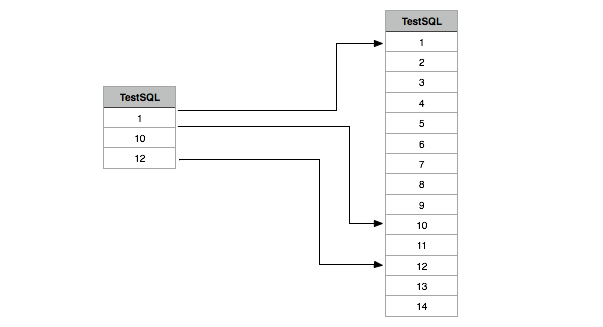
select \* from TestSQL where UserID >1 and UserID < 5 order by UserID desc,ArriveDate desc

## join 中使用索引

我们来新建一个TestSQL\_join表，其结构与TestSQL 相同，但只有Mobile索引。首先来看一个简单的join操作：

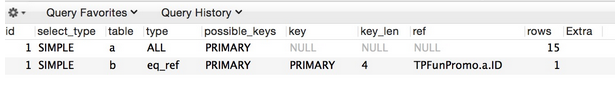
select \* from TestSQL join TestSQL\_join on TestSQL.id = TestSQL.id

MySQL首先比较TestSQL和 TestSQL\_join表那个行数少，如TestSQL中的记录较少，TestSQL就是一个小表，而TestSQL\_join则是大表，MySQL引擎先把TestSQL中的ID全部去出来，然后根据id到TestSQL\_join中查询相关的记录。

 在这里，TestSQL中行数决定了循环的次数，但是TestSQL\_join则决定了每次循环查询所需要查询的时间；这时如果TestSQL\_join中的ID是索引则会大大减少查询时间如下SQL:

select \* from TestSQL a join TestSQL\_join b on a.ID = b.ID

由于b.ID 是TestSQL\_join的主键，查询使用了TestSQL\_join主键索引。

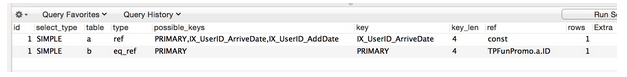


如果此时对TestSQL增加条件筛选：

select \* from TestSQL a join TestSQL\_join b on a.ID = b.ID

where a.UserID = 1;

因为a.UserID 是TestSQL的索引，所以在过滤TestSQL表的行数时，采用次索引查询对应ID，然后根据ID查询TestSQL\_join的记录。



如果此时针对TestSQL\_join 增加where条件过滤：

select \* from TestSQL a join TestSQL\_join b on a.ID = b.ID

where a.UserID >1 and b.Mobile = "2147483647"

这种时候，因为TestSQL 和 TestSQL\_join 根据where条件所筛选出来的行数大小可能会有变化，也就是说TestSQL\_join 有可能会变成小表，这时候将会优先从TestSQL\_join 查询出相关ID，然后根据ID去查询TestSQL。

join操作时，大表小表的概念，主要是按照两张表分别执行对应查询条件，哪个开销更小，哪个就是小表。join操作虽然在SQL层面很方便，而且在线上大流量的情况下，一旦SQL的join操作导致查询缓慢，较难即使优化。另外在服务化的系统中，容易导致业务领域不清晰，所以在互联网大流量的应用中是不推荐使用join操作的。

# 索引建立的原则

* 使用区分度高的列作为索引
* 区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，区分度越高，索引树的分叉也就越多，一次性找到的概率也就越高。
* 尽量使用字段长度小的列作为索引
* 使用数据类型简单的列（int 型，固定长度）
* 选用NOT NULL的列
* 在MySQL中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值。
* 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。这样也可避免索引重复。

# 索引使用的原则

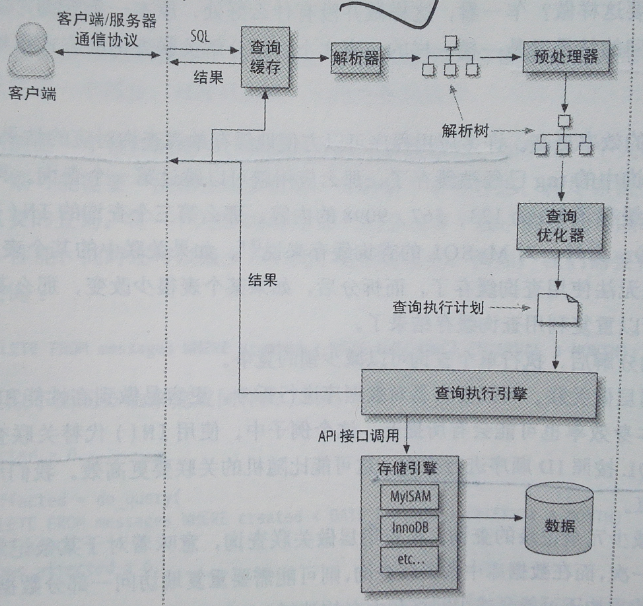
最左前缀匹配原则(leftmost)，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，=和in可以乱序，一个联合索引中，如UserID，ArriveDate的联合索引，使用ArriveDate in （）and UserID = 的任意顺序，MySQL的查询优化器会帮你优化成索引可以识别的形式

索引列不能参与计算：select \* from TestSQL where UserID + 1 >1 and UserID < 5 这种方式UserID 的索引就不会再被使用，因为在进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。

另外当使用<>,like通配符放置在最前面 如：like’%ddd’ ,not, in, !=等运算符都不会使用索引。

查询数据库记录时，查询到的条目数尽量小，当通过索引获取到的数据库记录> 数据库总揭露的1/3时，SQL将有可能直接全表扫描，索引就失去了应有的作用。

# 查询优化解析



注意：mysql多表关联是使用嵌套循环，回溯所有表的关联

## 优化count

count(\*):它会忽略所有的列而直接他估计所有的行数。

## 优化group by 和distinct(通过索引优化)

使用索引来进行优化。另外，尽量使用group by 来代替distinct来去重。

## 优化limit

使用 索引覆盖扫描：

|  |
| --- |
| select film\_id,description from order by title limit 50,5; |
| #使用 索引覆盖扫描  select film.film\_id, film.description from film  join (  select film\_id from film order by title limit 50,5) as lim  on film.film\_id=lim.film\_id |

## 优化union

MySQL总是通过创建并填充临时表的方式来执行UNION查询。可以将where,limit,orderby等子句下推到UNION的各个子查询中，一边优化器可以充分利用这些条件优化。

|  |
| --- |
| select first\_name,last\_name from actor order by last\_name limit 20  union all (Select first\_name,last\_name  from actor  order by last\_name  limit 20)  limit 20;  #除非程序确实需要消除重复行，否则一定要使用UNION ALL |

## 优化排名语句

|  |
| --- |
| set @curr\_cnt:=0, @prev\_cnt:=0, @rank:=0  select actor\_id,  @curr\_cnt:=cnt as cnt,  @rank := if(@prev\_cnt <> @curr\_cnt,@rank+1,@rank) as rank,  @prev\_cnt :=@curr\_cnt as dummy  from (  SELECT actor\_id,count(\*) as cnt  From film\_actor  Group by actor\_id  Order by cnt desc  Limit 10  )as der; |

## 统计更新和插入的数量

|  |
| --- |
| Insert into t1(c1,c2) values (4,4),(2,1),(3,1)  On duplicate key update c1=values(c1) + (0\*(@x:=@x+1));  #@x表示更新数 |