**MySQL索引原理及慢查询优化**

MySQL凭借着出色的性能、低廉的成本、丰富的资源，已经成为绝大多数互联网公司的首选关系型数据库。虽然性能出色，但所谓“好马配好鞍”，如何能够更好的使用它，已经成为开发工程师的必修课，我们经常会从职位描述上看到诸如“精通MySQL”、“SQL语句优化”、“了解数据库原理”等要求。我们知道一般的应用系统，读写比例在10:1左右，而且插入操作和一般的更新操作很少出现性能问题，遇到最多的，也是最容易出问题的，还是一些复杂的查询操作，所以查询语句的优化显然是重中之重。  
本人从13年7月份起，一直在美团核心业务系统部做慢查询的优化工作，共计十余个系统，累计解决和积累了上百个慢查询案例。随着业务的复杂性提升，遇到的问题千奇百怪，五花八门，匪夷所思。本文旨在以开发工程师的角度来解释数据库索引的原理和如何优化慢查询。

# 一个慢查询引发的思考

select

count(\*)

from

task

where

status=2

and operator\_id=20839

and operate\_time>1371169729

and operate\_time<1371174603

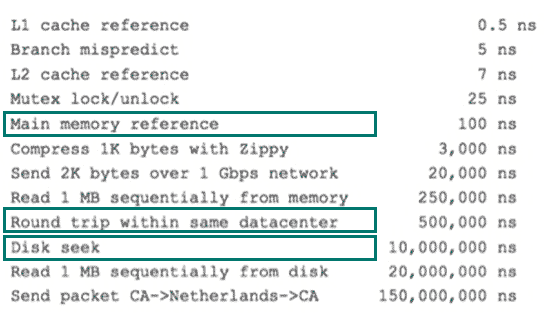
and type=2;

系统使用者反应有一个功能越来越慢，于是工程师找到了上面的SQL。  
并且兴致冲冲的找到了我，“这个SQL需要优化，给我把每个字段都加上索引”  
我很惊讶，问道“为什么需要每个字段都加上索引？”  
“把查询的字段都加上索引会更快”工程师信心满满  
“这种情况完全可以建一个联合索引，因为是最左前缀匹配，所以operate\_time需要放到最后，而且还需要把其他相关的查询都拿来，需要做一个综合评估。”  
“联合索引？最左前缀匹配？综合评估？”工程师不禁陷入了沉思。  
多数情况下，我们知道索引能够提高查询效率，但应该如何建立索引？索引的顺序如何？许多人却只知道大概。其实理解这些概念并不难，而且索引的原理远没有想象的那么复杂。

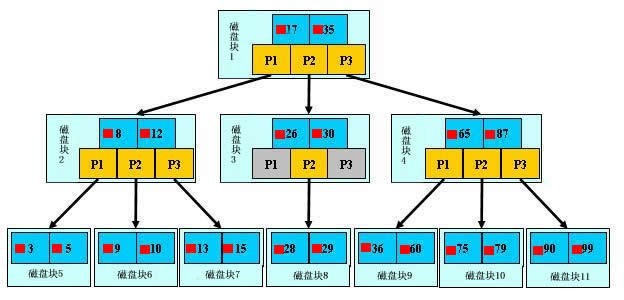
# MySQL索引原理

##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？是不是觉得如果没有索引，这个事情根本无法完成？

##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。  
数据库也是一样，但显然要复杂许多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等等。数据库应该选择怎么样的方式来应对所有的问题呢？我们回想字典的例子，能不能把数据分成段，然后分段查询呢？最简单的如果1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段......这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。但如果是1千万的记录呢，分成几段比较好？稍有算法基础的同学会想到搜索树，其平均复杂度是lgN，具有不错的查询性能。但这里我们忽略了一个关键的问题，复杂度模型是基于每次相同的操作成本来考虑的，数据库实现比较复杂，数据保存在磁盘上，而为了提高性能，每次又可以把部分数据读入内存来计算，因为我们知道访问磁盘的成本大概是访问内存的十万倍左右，所以简单的搜索树难以满足复杂的应用场景。

###磁盘IO与预读  
前面提到了访问磁盘，那么这里先简单介绍一下磁盘IO和预读，磁盘读取数据靠的是机械运动，每次读取数据花费的时间可以分为寻道时间、旋转延迟、传输时间三个部分，寻道时间指的是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下；旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是1/120/2 = 4.17ms；传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。那么访问一次磁盘的时间，即一次磁盘IO的时间约等于5+4.17 = 9ms左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十万百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。下图是计算机硬件延迟的对比图，供大家参考：  
  
考虑到磁盘IO是非常高昂的操作，计算机操作系统做了一些优化，当一次IO时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候，实际上才发生了一次IO，这个理论对于索引的数据结构设计非常有帮助。

###索引的数据结构  
前面讲了生活中索引的例子，索引的基本原理，数据库的复杂性，又讲了操作系统的相关知识，目的就是让大家了解，任何一种数据结构都不是凭空产生的，一定会有它的背景和使用场景，我们现在总结一下，我们需要这种数据结构能够做些什么，其实很简单，那就是：每次查找数据时把磁盘IO次数控制在一个很小的数量级，最好是常数数量级。那么我们就想到如果一个高度可控的多路搜索树是否能满足需求呢？就这样，b+树应运而生。

###详解b+树  
  
如上图，是一颗b+树，关于b+树的定义可以参见[B+树](http://zh.wikipedia.org/wiki/B%2B%E6%A0%91)，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

###b+树的查找过程  
如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

###b+树性质  
1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。  
2.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

# 慢查询优化

关于MySQL索引原理是比较枯燥的东西，大家只需要有一个感性的认识，并不需要理解得非常透彻和深入。我们回头来看看一开始我们说的慢查询，了解完索引原理之后，大家是不是有什么想法呢？先总结一下索引的几大基本原则

## 建索引的几大原则

1.最左前缀匹配原则，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.=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式  
3.尽量选择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，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录  
4.索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);  
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

## 回到开始的慢查询

根据最左匹配原则，最开始的sql语句的索引应该是status、operator\_id、type、operate\_time的联合索引；其中status、operator\_id、type的顺序可以颠倒，所以我才会说，把这个表的所有相关查询都找到，会综合分析；  
比如还有如下查询

select \* from task where status = 0 and type = 12 limit 10;

select count(\*) from task where status = 0 ;

那么索引建立成(status,type,operator\_id,operate\_time)就是非常正确的，因为可以覆盖到所有情况。这个就是利用了索引的最左匹配的原则

## 查询优化神器 - explain命令

关于explain命令相信大家并不陌生，具体用法和字段含义可以参考官网[explain-output](http://dev.mysql.com/doc/refman/5.5/en/explain-output.html)，这里需要强调rows是核心指标，绝大部分rows小的语句执行一定很快（有例外，下面会讲到）。所以优化语句基本上都是在优化rows。

## 慢查询优化基本步骤

0.先运行看看是否真的很慢，注意设置SQL\_NO\_CACHE  
1.where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高  
2.explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）  
3.order by limit 形式的sql语句让排序的表优先查  
4.了解业务方使用场景  
5.加索引时参照建索引的几大原则  
6.观察结果，不符合预期继续从0分析

## 几个慢查询案例

下面几个例子详细解释了如何分析和优化慢查询

### 复杂语句写法

很多情况下，我们写SQL只是为了实现功能，这只是第一步，不同的语句书写方式对于效率往往有本质的差别，这要求我们对mysql的执行计划和索引原则有非常清楚的认识，请看下面的语句

select

distinct cert.emp\_id

from

cm\_log cl

inner join

(

select

emp.id as emp\_id,

emp\_cert.id as cert\_id

from

employee emp

left join

emp\_certificate emp\_cert

on emp.id = emp\_cert.emp\_id

where

emp.is\_deleted=0

) cert

on (

cl.ref\_table='Employee'

and cl.ref\_oid= cert.emp\_id

)

or (

cl.ref\_table='EmpCertificate'

and cl.ref\_oid= cert.cert\_id

)

where

cl.last\_upd\_date >='2013-11-07 15:03:00'

and cl.last\_upd\_date<='2013-11-08 16:00:00';

0.先运行一下，53条记录 1.87秒，又没有用聚合语句，比较慢

53 rows in set (1.87 sec)

1.explain

+----+-------------+------------+-------+---------------------------------+-----------------------+---------+-------------------+-------+--------------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+------------+-------+---------------------------------+-----------------------+---------+-------------------+-------+--------------------------------+

| 1 | PRIMARY | cl | range | cm\_log\_cls\_id,idx\_last\_upd\_date | idx\_last\_upd\_date | 8 | NULL | 379 | Using where; Using temporary |

| 1 | PRIMARY | <derived2> | ALL | NULL | NULL | NULL | NULL | 63727 | Using where; Using join buffer |

| 2 | DERIVED | emp | ALL | NULL | NULL | NULL | NULL | 13317 | Using where |

| 2 | DERIVED | emp\_cert | ref | emp\_certificate\_empid | emp\_certificate\_empid | 4 | meituanorg.emp.id | 1 | Using index |

+----+-------------+------------+-------+---------------------------------+-----------------------+---------+-------------------+-------+--------------------------------+

简述一下执行计划，首先mysql根据idx\_last\_upd\_date索引扫描cm\_log表获得379条记录；然后查表扫描了63727条记录，分为两部分，derived表示构造表，也就是不存在的表，可以简单理解成是一个语句形成的结果集，后面的数字表示语句的ID。derived2表示的是ID = 2的查询构造了虚拟表，并且返回了63727条记录。我们再来看看ID = 2的语句究竟做了写什么返回了这么大量的数据，首先全表扫描employee表13317条记录，然后根据索引emp\_certificate\_empid关联emp\_certificate表，rows = 1表示，每个关联都只锁定了一条记录，效率比较高。获得后，再和cm\_log的379条记录根据规则关联。从执行过程上可以看出返回了太多的数据，返回的数据绝大部分cm\_log都用不到，因为cm\_log只锁定了379条记录。  
如何优化呢？可以看到我们在运行完后还是要和cm\_log做join,那么我们能不能之前和cm\_log做join呢？仔细分析语句不难发现，其基本思想是如果cm\_log的ref\_table是EmpCertificate就关联emp\_certificate表，如果ref\_table是Employee就关联employee表，我们完全可以拆成两部分，并用union连接起来，注意这里用union，而不用union all是因为原语句有“distinct”来得到唯一的记录，而union恰好具备了这种功能。如果原语句中没有distinct不需要去重，我们就可以直接使用union all了，因为使用union需要去重的动作，会影响SQL性能。  
优化过的语句如下

select

emp.id

from

cm\_log cl

inner join

employee emp

on cl.ref\_table = 'Employee'

and cl.ref\_oid = emp.id

where

cl.last\_upd\_date >='2013-11-07 15:03:00'

and cl.last\_upd\_date<='2013-11-08 16:00:00'

and emp.is\_deleted = 0

union

select

emp.id

from

cm\_log cl

inner join

emp\_certificate ec

on cl.ref\_table = 'EmpCertificate'

and cl.ref\_oid = ec.id

inner join

employee emp

on emp.id = ec.emp\_id

where

cl.last\_upd\_date >='2013-11-07 15:03:00'

and cl.last\_upd\_date<='2013-11-08 16:00:00'

and emp.is\_deleted = 0

4.不需要了解业务场景，只需要改造的语句和改造之前的语句保持结果一致

5.现有索引可以满足，不需要建索引

6.用改造后的语句实验一下，只需要10ms 降低了近200倍！

+----+--------------+------------+--------+---------------------------------+-------------------+---------+-----------------------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+--------------+------------+--------+---------------------------------+-------------------+---------+-----------------------+------+-------------+

| 1 | PRIMARY | cl | range | cm\_log\_cls\_id,idx\_last\_upd\_date | idx\_last\_upd\_date | 8 | NULL | 379 | Using where |

| 1 | PRIMARY | emp | eq\_ref | PRIMARY | PRIMARY | 4 | meituanorg.cl.ref\_oid | 1 | Using where |

| 2 | UNION | cl | range | cm\_log\_cls\_id,idx\_last\_upd\_date | idx\_last\_upd\_date | 8 | NULL | 379 | Using where |

| 2 | UNION | ec | eq\_ref | PRIMARY,emp\_certificate\_empid | PRIMARY | 4 | meituanorg.cl.ref\_oid | 1 | |

| 2 | UNION | emp | eq\_ref | PRIMARY | PRIMARY | 4 | meituanorg.ec.emp\_id | 1 | Using where |

| NULL | UNION RESULT | <union1,2> | ALL | NULL | NULL | NULL | NULL | NULL | |

+----+--------------+------------+--------+---------------------------------+-------------------+---------+-----------------------+------+-------------+

53 rows in set (0.01 sec)

### 明确应用场景

举这个例子的目的在于颠覆我们对列的区分度的认知，一般上我们认为区分度越高的列，越容易锁定更少的记录，但在一些特殊的情况下，这种理论是有局限性的

select

\*

from

stage\_poi sp

where

sp.accurate\_result=1

and (

sp.sync\_status=0

or sp.sync\_status=2

or sp.sync\_status=4

);

0.先看看运行多长时间,951条数据6.22秒，真的很慢

951 rows in set (6.22 sec)

1.先explain，rows达到了361万，type = ALL表明是全表扫描

+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+

| 1 | SIMPLE | sp | ALL | NULL | NULL | NULL | NULL | 3613155 | Using where |

+----+-------------+-------+------+---------------+------+---------+------+---------+-------------+

2.所有字段都应用查询返回记录数，因为是单表查询 0已经做过了951条

3.让explain的rows 尽量逼近951

看一下accurate\_result = 1的记录数

select count(\*),accurate\_result from stage\_poi group by accurate\_result;

+----------+-----------------+

| count(\*) | accurate\_result |

+----------+-----------------+

| 1023 | -1 |

| 2114655 | 0 |

| 972815 | 1 |

+----------+-----------------+

我们看到accurate\_result这个字段的区分度非常低，整个表只有-1,0,1三个值，加上索引也无法锁定特别少量的数据

再看一下sync\_status字段的情况

select count(\*),sync\_status from stage\_poi group by sync\_status;

+----------+-------------+

| count(\*) | sync\_status |

+----------+-------------+

| 3080 | 0 |

| 3085413 | 3 |

+----------+-------------+

同样的区分度也很低，根据理论，也不适合建立索引

问题分析到这，好像得出了这个表无法优化的结论，两个列的区分度都很低，即便加上索引也只能适应这种情况，很难做普遍性的优化，比如当sync\_status 0、3分布的很平均，那么锁定记录也是百万级别的

4.找业务方去沟通，看看使用场景。业务方是这么来使用这个SQL语句的，每隔五分钟会扫描符合条件的数据，处理完成后把sync\_status这个字段变成1,五分钟符合条件的记录数并不会太多，1000个左右。了解了业务方的使用场景后，优化这个SQL就变得简单了，因为业务方保证了数据的不平衡，如果加上索引可以过滤掉绝大部分不需要的数据

5.根据建立索引规则，使用如下语句建立索引

alter table stage\_poi add index idx\_acc\_status(accurate\_result,sync\_status);

6.观察预期结果,发现只需要200ms，快了30多倍。

952 rows in set (0.20 sec)

我们再来回顾一下分析问题的过程，单表查询相对来说比较好优化，大部分时候只需要把where条件里面的字段依照规则加上索引就好，如果只是这种“无脑”优化的话，显然一些区分度非常低的列，不应该加索引的列也会被加上索引，这样会对插入、更新性能造成严重的影响，同时也有可能影响其它的查询语句。所以我们第4步调差SQL的使用场景非常关键，我们只有知道这个业务场景，才能更好地辅助我们更好的分析和优化查询语句。

### 无法优化的语句

select

c.id,

c.name,

c.position,

c.sex,

c.phone,

c.office\_phone,

c.feature\_info,

c.birthday,

c.creator\_id,

c.is\_keyperson,

c.giveup\_reason,

c.status,

c.data\_source,

from\_unixtime(c.created\_time) as created\_time,

from\_unixtime(c.last\_modified) as last\_modified,

c.last\_modified\_user\_id

from

contact c

inner join

contact\_branch cb

on c.id = cb.contact\_id

inner join

branch\_user bu

on cb.branch\_id = bu.branch\_id

and bu.status in (

1,

2)

inner join

org\_emp\_info oei

on oei.data\_id = bu.user\_id

and oei.node\_left >= 2875

and oei.node\_right <= 10802

and oei.org\_category = - 1

order by

c.created\_time desc limit 0 ,

10;

还是几个步骤  
0.先看语句运行多长时间，10条记录用了13秒，已经不可忍受

10 rows in set (13.06 sec)

1.explain

+----+-------------+-------+--------+-------------------------------------+-------------------------+---------+--------------------------+------+----------------------------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+--------+-------------------------------------+-------------------------+---------+--------------------------+------+----------------------------------------------+

| 1 | SIMPLE | oei | ref | idx\_category\_left\_right,idx\_data\_id | idx\_category\_left\_right | 5 | const | 8849 | Using where; Using temporary; Using filesort |

| 1 | SIMPLE | bu | ref | PRIMARY,idx\_userid\_status | idx\_userid\_status | 4 | meituancrm.oei.data\_id | 76 | Using where; Using index |

| 1 | SIMPLE | cb | ref | idx\_branch\_id,idx\_contact\_branch\_id | idx\_branch\_id | 4 | meituancrm.bu.branch\_id | 1 | |

| 1 | SIMPLE | c | eq\_ref | PRIMARY | PRIMARY | 108 | meituancrm.cb.contact\_id | 1 | |

+----+-------------+-------+--------+-------------------------------------+-------------------------+---------+--------------------------+------+----------------------------------------------+

从执行计划上看，mysql先查org\_emp\_info表扫描8849记录，再用索引idx\_userid\_status关联branch\_user表，再用索引idx\_branch\_id关联contact\_branch表，最后主键关联contact表。  
rows返回的都非常少，看不到有什么异常情况。我们在看一下语句，发现后面有order by + limit组合，会不会是排序量太大搞的？于是我们简化SQL，去掉后面的order by 和 limit，看看到底用了多少记录来排序

select

count(\*)

from

contact c

inner join

contact\_branch cb

on c.id = cb.contact\_id

inner join

branch\_user bu

on cb.branch\_id = bu.branch\_id

and bu.status in (

1,

2)

inner join

org\_emp\_info oei

on oei.data\_id = bu.user\_id

and oei.node\_left >= 2875

and oei.node\_right <= 10802

and oei.org\_category = - 1

+----------+

| count(\*) |

+----------+

| 778878 |

+----------+

1 row in set (5.19 sec)

发现排序之前居然锁定了778878条记录，如果针对70万的结果集排序，将是灾难性的，怪不得这么慢，那我们能不能换个思路，先根据contact的created\_time排序，再来join会不会比较快呢？  
于是改造成下面的语句，也可以用straight\_join来优化  
select  
c.id,  
c.name,  
c.position,  
c.sex,  
c.phone,  
c.office\_phone,  
c.feature\_info,  
c.birthday,  
c.creator\_id,  
c.is\_keyperson,  
c.giveup\_reason,  
c.status,  
c.data\_source,  
from\_unixtime(c.created\_time) as created\_time,  
from\_unixtime(c.last\_modified) as last\_modified,  
c.last\_modified\_user\_id  
from  
contact c  
where  
exists (  
select  
1  
from  
contact\_branch cb  
inner join  
branch\_user bu  
on cb.branch\_id = bu.branch\_id  
and bu.status in (  
1,  
2)  
inner join  
org\_emp\_info oei  
on oei.data\_id = bu.user\_id  
and oei.node\_left >= 2875  
and oei.node\_right <= 10802  
and oei.org\_category = - 1  
where  
c.id = cb.contact\_id  
)  
order by  
c.created\_time desc limit 0 ,  
10;

验证一下效果 预计在1ms内，提升了13000多倍！

```sql

10 rows in set (0.00 sec)

本以为至此大工告成，但我们在前面的分析中漏了一个细节，先排序再join和先join再排序理论上开销是一样的，为何提升这么多是因为有一个limit！大致执行过程是：mysql先按索引排序得到前10条记录，然后再去join过滤，当发现不够10条的时候，再次去10条，再次join，这显然在内层join过滤的数据非常多的时候，将是灾难的，极端情况，内层一条数据都找不到，mysql还傻乎乎的每次取10条，几乎遍历了这个数据表！  
用不同参数的SQL试验下

select

sql\_no\_cache c.id,

c.name,

c.position,

c.sex,

c.phone,

c.office\_phone,

c.feature\_info,

c.birthday,

c.creator\_id,

c.is\_keyperson,

c.giveup\_reason,

c.status,

c.data\_source,

from\_unixtime(c.created\_time) as created\_time,

from\_unixtime(c.last\_modified) as last\_modified,

c.last\_modified\_user\_id

from

contact c

where

exists (

select

1

from

contact\_branch cb

inner join

branch\_user bu

on cb.branch\_id = bu.branch\_id

and bu.status in (

1,

2)

inner join

org\_emp\_info oei

on oei.data\_id = bu.user\_id

and oei.node\_left >= 2875

and oei.node\_right <= 2875

and oei.org\_category = - 1

where

c.id = cb.contact\_id

)

order by

c.created\_time desc limit 0 ,

10;

Empty set (2 min 18.99 sec)

2 min 18.99 sec！比之前的情况还糟糕很多。由于mysql的nested loop机制，遇到这种情况，基本是无法优化的。这条语句最终也只能交给应用系统去优化自己的逻辑了。  
通过这个例子我们可以看到，并不是所有语句都能优化，而往往我们优化时，由于SQL用例回归时落掉一些极端情况，会造成比原来还严重的后果。所以，第一：不要指望所有语句都能通过SQL优化，第二：不要过于自信，只针对具体case来优化，而忽略了更复杂的情况。

慢查询的案例就分析到这儿，以上只是一些比较典型的案例。我们在优化过程中遇到过超过1000行，涉及到16个表join的“垃圾SQL”，也遇到过线上线下数据库差异导致应用直接被慢查询拖死，也遇到过varchar等值比较没有写单引号，还遇到过笛卡尔积查询直接把从库搞死。再多的案例其实也只是一些经验的积累，如果我们熟悉查询优化器、索引的内部原理，那么分析这些案例就变得特别简单了。

# 写在后面的话

本文以一个慢查询案例引入了MySQL索引原理、优化慢查询的一些方法论;并针对遇到的典型案例做了详细的分析。其实做了这么长时间的语句优化后才发现，任何数据库层面的优化都抵不上应用系统的优化，同样是MySQL，可以用来支撑Google/FaceBook/Taobao应用，但可能连你的个人网站都撑不住。

# Mysql----索引实现剖析与实践

# 摘要

本文以[**MySQL数据库**](http://lib.csdn.net/base/mysql)为研究对象，讨论与数据库索引相关的一些话题。特别需要说明的是，MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等。为了避免混乱，本文将只关注于BTree索引，因为这是平常使用MySQL时主要打交道的索引，至于哈希索引和全文索引本文暂不讨论。

文章主要内容分为三个部分。

第一部分主要从[**数据结构**](http://lib.csdn.net/base/datastructure)及[**算法**](http://lib.csdn.net/base/datastructure)理论层面讨论MySQL数据库索引的数理基础。

第二部分结合MySQL数据库中MyISAM和InnoDB数据存储引擎中索引的[**架构**](http://lib.csdn.net/base/architecture)实现讨论聚集索引、非聚集索引及覆盖索引等话题。

第三部分根据上面的理论基础，讨论MySQL中高性能使用索引的策略。

# 数据结构及算法基础

## 索引的本质

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。提取句子主干，就可以得到索引的本质：索引是数据结构。

我们知道，数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。最基本的查询算法当然是[顺序查找](http://en.wikipedia.org/wiki/Linear_search)（linear search），这种复杂度为O(n)的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如[二分查找](http://en.wikipedia.org/wiki/Binary_search_algorithm)（binary search）、[二叉树查找](http://en.wikipedia.org/wiki/Binary_search_tree)（binary tree search）等。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于[二叉查找树](http://en.wikipedia.org/wiki/Binary_search_tree)上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

看一个例子：

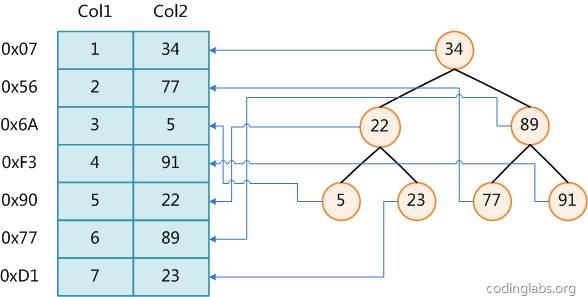


图1

图1展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)O(log2n)的复杂度内获取到相应数据。

虽然这是一个货真价实的索引，但是实际的数据库系统几乎没有使用二叉查找树或其进化品种[红黑树](http://en.wikipedia.org/wiki/Red-black_tree)（red-black tree）实现的，原因会在下文介绍。

## B-Tree和B+Tree

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构，在本文的下一节会结合存储器原理及计算机存取原理讨论为什么B-Tree和B+Tree在被如此广泛用于索引，这一节先单纯从数据结构角度描述它们。

### B-Tree

为了描述B-Tree，首先定义一条数据记录为一个二元组[key, data]，key为记录的键值，对于不同数据记录，key是互不相同的；data为数据记录除key外的数据。那么B-Tree是满足下列条件的数据结构：

d为大于1的一个正整数，称为B-Tree的度。

h为一个正整数，称为B-Tree的高度。

每个非叶子节点由n-1个key和n个指针组成，其中d<=n<=2d。

每个叶子节点最少包含一个key和两个指针，最多包含2d-1个key和2d个指针，叶节点的指针均为null 。

所有叶节点具有相同的深度，等于树高h。

key和指针互相间隔，节点两端是指针。

一个节点中的key从左到右非递减排列。

所有节点组成树结构。

每个指针要么为null，要么指向另外一个节点。

如果某个指针在节点node最左边且不为null，则其指向节点的所有key小于v(key1)v(key1)，其中v(key1)v(key1)为node的第一个key的值。

如果某个指针在节点node最右边且不为null，则其指向节点的所有key大于v(keym)v(keym)，其中v(keym)v(keym)为node的最后一个key的值。

如果某个指针在节点node的左右相邻key分别是keyikeyi和keyi+1keyi+1且不为null，则其指向节点的所有key小于v(keyi+1)v(keyi+1)且大于v(keyi)v(keyi)。

图2是一个d=2的B-Tree示意图。

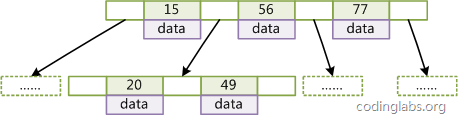


图2

由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

1. BTree\_Search(node, key) {
2. if(node == null) return null;
3. foreach(node.key)
4. {
5. if(node.key[i] == key) return node.data[i];
6. if(node.key[i] > key) return BTree\_Search(point[i]->node);
7. }
8. return BTree\_Search(point[i+1]->node);
9. }
10. data = BTree\_Search(root, my\_key);

关于B-Tree有一系列有趣的性质，例如一个度为d的B-Tree，设其索引N个key，则其树高h的上限为logd((N+1)/2)logd((N+1)/2)，检索一个key，其查找节点个数的渐进复杂度为O(logdN)O(logdN)。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

另外，由于插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质，本文不打算完整讨论B-Tree这些内容，因为已经有许多资料详细说明了B-Tree的数学性质及插入删除算法，有兴趣的朋友可以在本文末的参考文献一栏找到相应的资料进行阅读。

### B+Tree

B-Tree有许多变种，其中最常见的是B+Tree，例如MySQL就普遍使用B+Tree实现其索引结构。

与B-Tree相比，B+Tree有以下不同点：

每个节点的指针上限为2d而不是2d+1。

内节点不存储data，只存储key；叶子节点不存储指针。

图3是一个简单的B+Tree示意。



图3

由于并不是所有节点都具有相同的域，因此B+Tree中叶节点和内节点一般大小不同。这点与B-Tree不同，虽然B-Tree中不同节点存放的key和指针可能数量不一致，但是每个节点的域和上限是一致的，所以在实现中B-Tree往往对每个节点申请同等大小的空间。

一般来说，B+Tree比B-Tree更适合实现外存储索引结构，具体原因与外存储器原理及计算机存取原理有关，将在下面讨论。

### 带有顺序访问指针的B+Tree

一般在数据库系统或文件系统中使用的B+Tree结构都在经典B+Tree的基础上进行了优化，增加了顺序访问指针。

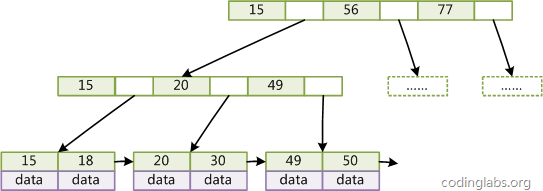


图4

如图4所示，在B+Tree的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能，例如图4中如果要查询key为从18到49的所有数据记录，当找到18后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提到了区间查询效率。

这一节对B-Tree和B+Tree进行了一个简单的介绍，下一节结合存储器存取原理介绍为什么目前B+Tree是数据库系统实现索引的首选数据结构。

## 为什么使用B-Tree（B+Tree）

上文说过，红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。下面先介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

### 主存存取原理

目前计算机使用的主存基本都是随机读写存储器（RAM），现代RAM的结构和存取原理比较复杂，这里本文抛却具体差别，抽象出一个十分简单的存取模型来说明RAM的工作原理。

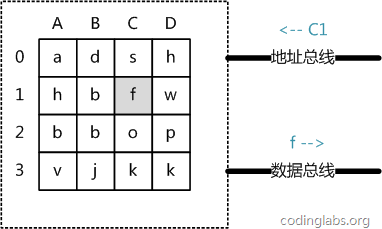


图5

从抽象角度看，主存是一系列的存储单元组成的矩阵，每个存储单元存储固定大小的数据。每个存储单元有唯一的地址，现代主存的编址规则比较复杂，这里将其简化成一个二维地址：通过一个行地址和一个列地址可以唯一定位到一个存储单元。图5展示了一个4 x 4的主存模型。

主存的存取过程如下：

当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。

写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

这里可以看出，主存存取的时间仅与存取次数呈线性关系，因为不存在机械操作，两次存取的数据的“距离”不会对时间有任何影响，例如，先取A0再取A1和先取A0再取D3的时间消耗是一样的。

### 磁盘存取原理

上文说过，索引一般以文件形式存储在磁盘上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。

图6是磁盘的整体结构示意图。

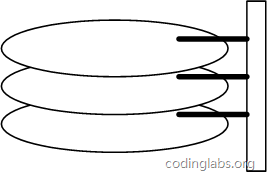


图6

一个磁盘由大小相同且同轴的圆形盘片组成，磁盘可以转动（各个磁盘必须同步转动）。在磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不能转动，但是可以沿磁盘半径方向运动（实际是斜切向运动），每个磁头同一时刻也必须是同轴的，即从正上方向下看，所有磁头任何时候都是重叠的（不过目前已经有多磁头独立技术，可不受此限制）。

图7是磁盘结构的示意图。

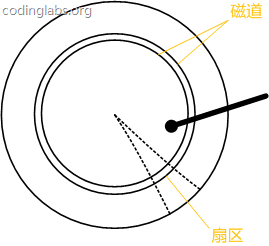


图7

盘片被划分成一系列同心环，圆心是盘片中心，每个同心环叫做一个磁道，所有半径相同的磁道组成一个柱面。磁道被沿半径线划分成一个个小的段，每个段叫做一个扇区，每个扇区是磁盘的最小存储单元。为了简单起见，我们下面假设磁盘只有一个盘片和一个磁头。

当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间。

### 局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及[**操作系统**](http://lib.csdn.net/base/operatingsystem)往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

### B-/+Tree索引的性能分析

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

上文还说过，B+Tree更适合外存索引，原因和内节点出度d有关。从上面分析可以看到，d越大索引的性能越好，而出度的上限取决于节点内key和data的大小：

dmax=floor(pagesize/(keysize+datasize+pointsize))dmax=floor(pagesize/(keysize+datasize+pointsize))

floor表示向下取整。由于B+Tree内节点去掉了data域，因此可以拥有更大的出度，拥有更好的性能。

这一章从理论角度讨论了与索引相关的数据结构与算法问题，下一章将讨论B+Tree是如何具体实现为MySQL中索引，同时将结合MyISAM和InnDB存储引擎介绍非聚集索引和聚集索引两种不同的索引实现形式。

# MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式

## MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：

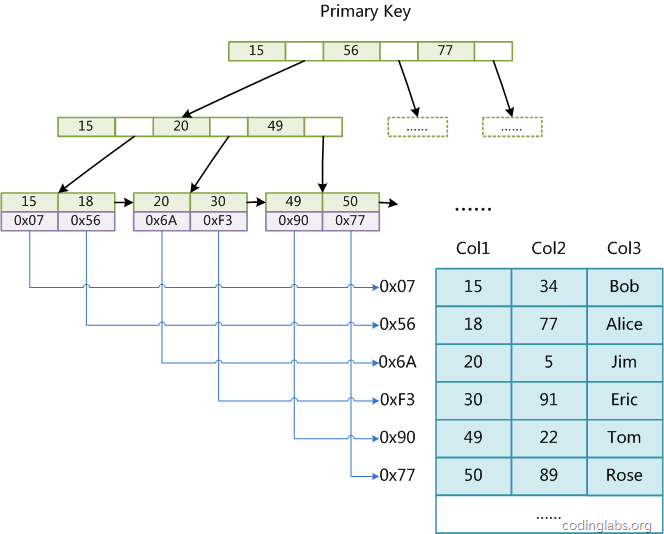


图8

这里设表一共有三列，假设我们以Col1为主键，则图8是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：

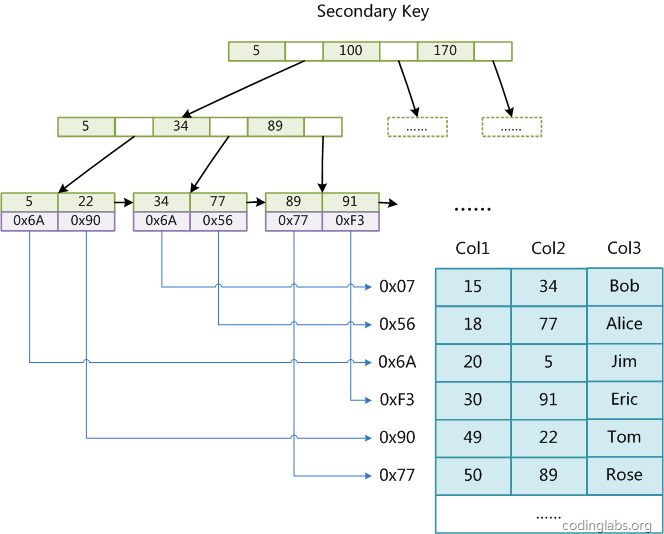


图9

同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分.

## InnoDB索引实现

虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



图10

图10是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，图11为定义在Col3上的一个辅助索引：

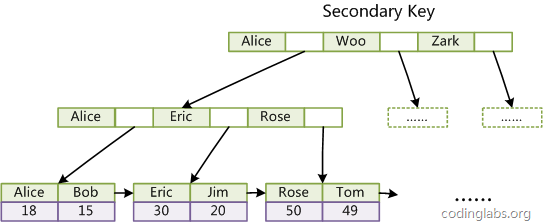


图11

这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

下一章将具体讨论这些与索引有关的优化策略。

# 索引使用策略及优化

MySQL的优化主要分为结构优化（Scheme optimization）和查询优化（Query optimization）。本章讨论的高性能索引策略主要属于结构优化范畴。本章的内容完全基于上文的理论基础，实际上一旦理解了索引背后的机制，那么选择高性能的策略就变成了纯粹的推理，并且可以理解这些策略背后的逻辑。

## 示例数据库

为了讨论索引策略，需要一个数据量不算小的数据库作为示例。本文选用MySQL官方文档中提供的示例数据库之一：employees。这个数据库关系复杂度适中，且数据量较大。下图是这个数据库的E-R关系图（引用自MySQL官方手册）：

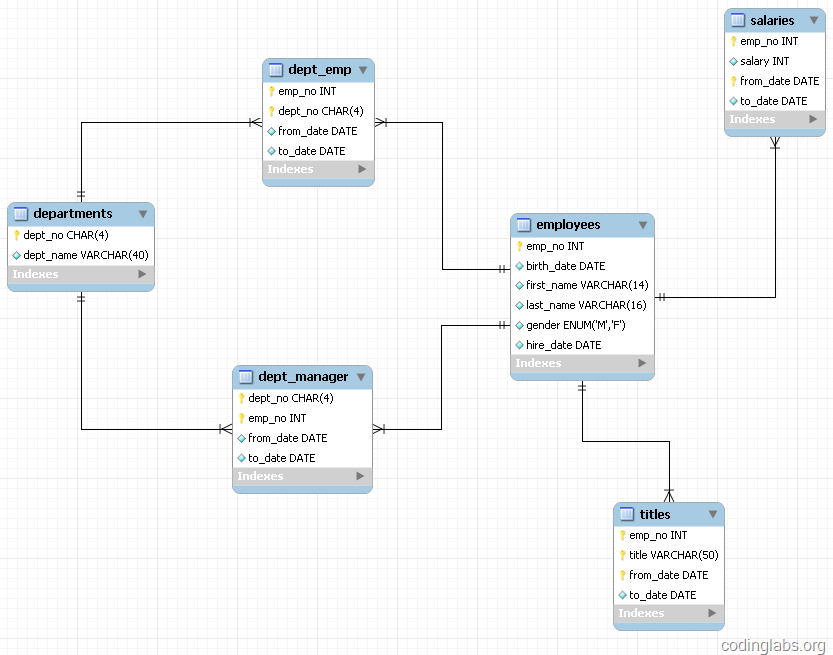


图12

MySQL官方文档中关于此数据库的页面为<http://dev.mysql.com/doc/employee/en/employee.html>。里面详细介绍了此数据库，并提供了下载地址和导入方法，如果有兴趣导入此数据库到自己的MySQL可以参考文中内容。

## 最左前缀原理与相关优化

高效使用索引的首要条件是知道什么样的查询会使用到索引，这个问题和B+Tree中的“最左前缀原理”有关，下面通过例子说明最左前缀原理。

这里先说一下联合索引的概念。在上文中，我们都是假设索引只引用了单个的列，实际上，MySQL中的索引可以以一定顺序引用多个列，这种索引叫做联合索引，一般的，一个联合索引是一个有序元组<a1, a2, …, an>，其中各个元素均为数据表的一列，实际上要严格定义索引需要用到关系代数，但是这里我不想讨论太多关系代数的话题，因为那样会显得很枯燥，所以这里就不再做严格定义。另外，单列索引可以看成联合索引元素数为1的特例。

以employees.titles表为例，下面先查看其上都有哪些索引：

1. SHOW INDEX FROM employees.titles;
2. +--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+
3. | Table | Non\_unique | Key\_name | Seq\_in\_index | Column\_name | Collation | Cardinality | Null | Index\_type |
4. +--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+
5. | titles | 0 | PRIMARY | 1 | emp\_no | A | NULL | | BTREE |
6. | titles | 0 | PRIMARY | 2 | title | A | NULL | | BTREE |
7. | titles | 0 | PRIMARY | 3 | from\_date | A | 443308 | | BTREE |
8. | titles | 1 | emp\_no | 1 | emp\_no | A | 443308 | | BTREE |
9. +--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+

从结果中可以到titles表的主索引为<emp\_no, title, from\_date>，还有一个辅助索引<emp\_no>。为了避免多个索引使事情变复杂（MySQL的SQL优化器在多索引时行为比较复杂），这里我们将辅助索引drop掉：

1. ALTER TABLE employees.titles DROP INDEX emp\_no;

这样就可以专心分析索引PRIMARY的行为了。

### 情况一：全列匹配。

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND title='Senior Engineer' AND from\_date='1986-06-26';
2. +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+
5. | 1 | SIMPLE | titles | const | PRIMARY | PRIMARY | 59 | const,const,const | 1 | |
6. +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+

很明显，当按照索引中所有列进行精确匹配（这里精确匹配指“=”或“IN”匹配）时，索引可以被用到。这里有一点需要注意，理论上索引对顺序是敏感的，但是由于MySQL的查询优化器会自动调整where子句的条件顺序以使用适合的索引，例如我们将where中的条件顺序颠倒：

1. EXPLAIN SELECT \* FROM employees.titles WHERE from\_date='1986-06-26' AND emp\_no='10001' AND title='Senior Engineer';
2. +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+
5. | 1 | SIMPLE | titles | const | PRIMARY | PRIMARY | 59 | const,const,const | 1 | |
6. +----+-------------+--------+-------+---------------+---------+---------+-------------------+------+-------+

效果是一样的。

### 情况二：最左前缀匹配。

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001';
2. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------+
5. | 1 | SIMPLE | titles | ref | PRIMARY | PRIMARY | 4 | const | 1 | |
6. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------+

当查询条件精确匹配索引的左边连续一个或几个列时，如<emp\_no>或<emp\_no, title>，所以可以被用到，但是只能用到一部分，即条件所组成的最左前缀。上面的查询从分析结果看用到了PRIMARY索引，但是key\_len为4，说明只用到了索引的第一列前缀。

### 情况三：查询条件用到了索引中列的精确匹配，但是中间某个条件未提供。

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND from\_date='1986-06-26';
2. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+
5. | 1 | SIMPLE | titles | ref | PRIMARY | PRIMARY | 4 | const | 1 | Using where |
6. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+

此时索引使用情况和情况二相同，因为title未提供，所以查询只用到了索引的第一列，而后面的from\_date虽然也在索引中，但是由于title不存在而无法和左前缀连接，因此需要对结果进行扫描过滤from\_date（这里由于emp\_no唯一，所以不存在扫描）。如果想让from\_date也使用索引而不是where过滤，可以增加一个辅助索引<emp\_no, from\_date>，此时上面的查询会使用这个索引。除此之外，还可以使用一种称之为“隔离列”的优化方法，将emp\_no与from\_date之间的“坑”填上。

首先我们看下title一共有几种不同的值：

1. SELECT DISTINCT(title) FROM employees.titles;
2. +--------------------+
3. | title |
4. +--------------------+
5. | Senior Engineer |
6. | Staff |
7. | Engineer |
8. | Senior Staff |
9. | Assistant Engineer |
10. | Technique Leader |
11. | Manager |
12. +--------------------+

只有7种。在这种成为“坑”的列值比较少的情况下，可以考虑用“IN”来填补这个“坑”从而形成最左前缀：

1. EXPLAIN SELECT \* FROM employees.titles
2. WHERE emp\_no='10001'
3. AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
4. AND from\_date='1986-06-26';
5. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
6. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
7. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
8. | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 59 | NULL | 7 | Using where |
9. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+

这次key\_len为59，说明索引被用全了，但是从type和rows看出IN实际上执行了一个range查询，这里检查了7个key。看下两种查询的性能比较：

1. SHOW PROFILES;
2. +----------+------------+-------------------------------------------------------------------------------+
3. | Query\_ID | Duration | Query |
4. +----------+------------+-------------------------------------------------------------------------------+
5. | 10 | 0.00058000 | SELECT \* FROM employees.titles WHERE emp\_no='10001' AND from\_date='1986-06-26'|
6. | 11 | 0.00052500 | SELECT \* FROM employees.titles WHERE emp\_no='10001' AND title IN ... |
7. +----------+------------+-------------------------------------------------------------------------------+

“填坑”后性能提升了一点。如果经过emp\_no筛选后余下很多数据，则后者性能优势会更加明显。当然，如果title的值很多，用填坑就不合适了，必须建立辅助索引。

### 情况四：查询条件没有指定索引第一列。

1. EXPLAIN SELECT \* FROM employees.titles WHERE from\_date='1986-06-26';
2. +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+
5. | 1 | SIMPLE | titles | ALL | NULL | NULL | NULL | NULL | 443308 | Using where |
6. +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+

由于不是最左前缀，索引这样的查询显然用不到索引。

### 情况五：匹配某列的前缀字符串。

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND title LIKE 'Senior%';
2. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
5. | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 56 | NULL | 1 | Using where |
6. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+

此时可以用到索引，（原文表述有误，如果通配符%不出现在开头，则可以用到索引，但根据具体情况不同可能只会用其中一个前缀）

### 情况六：范围查询。

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no < '10010' and title='Senior Engineer';
2. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
5. | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 4 | NULL | 16 | Using where |
6. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+

范围列可以用到索引（必须是最左前缀），但是范围列后面的列无法用到索引。同时，索引最多用于一个范围列，因此如果查询条件中有两个范围列则无法全用到索引。

1. EXPLAIN SELECT \* FROM employees.titles
2. WHERE emp\_no < '10010'
3. AND title='Senior Engineer'
4. AND from\_date BETWEEN '1986-01-01' AND '1986-12-31';
5. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
6. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
7. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
8. | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 4 | NULL | 16 | Using where |
9. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+

可以看到索引对第二个范围索引无能为力。这里特别要说明MySQL一个有意思的地方，那就是仅用explain可能无法区分范围索引和多值匹配，因为在type中这两者都显示为range。同时，用了“between”并不意味着就是范围查询，例如下面的查询：

1. EXPLAIN SELECT \* FROM employees.titles
2. WHERE emp\_no BETWEEN '10001' AND '10010'
3. AND title='Senior Engineer'
4. AND from\_date BETWEEN '1986-01-01' AND '1986-12-31';
5. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
6. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
7. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+
8. | 1 | SIMPLE | titles | range | PRIMARY | PRIMARY | 59 | NULL | 16 | Using where |
9. +----+-------------+--------+-------+---------------+---------+---------+------+------+-------------+

看起来是用了两个范围查询，但作用于emp\_no上的“BETWEEN”实际上相当于“IN”，也就是说emp\_no实际是多值精确匹配。可以看到这个查询用到了索引全部三个列。因此在MySQL中要谨慎地区分多值匹配和范围匹配，否则会对MySQL的行为产生困惑。

### 情况七：查询条件中含有函数或表达式。

很不幸，如果查询条件中含有函数或表达式，则MySQL不会为这列使用索引（虽然某些在数学意义上可以使用）。例如：

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no='10001' AND left(title, 6)='Senior';
2. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+
5. | 1 | SIMPLE | titles | ref | PRIMARY | PRIMARY | 4 | const | 1 | Using where |
6. +----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+

虽然这个查询和情况五中功能相同，但是由于使用了函数left，则无法为title列应用索引，而情况五中用LIKE则可以。再如：

1. EXPLAIN SELECT \* FROM employees.titles WHERE emp\_no - 1='10000';
2. +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+
5. | 1 | SIMPLE | titles | ALL | NULL | NULL | NULL | NULL | 443308 | Using where |
6. +----+-------------+--------+------+---------------+------+---------+------+--------+-------------+

显然这个查询等价于查询emp\_no为10001的函数，但是由于查询条件是一个表达式，MySQL无法为其使用索引。看来MySQL还没有智能到自动优化常量表达式的程度，因此在写查询语句时尽量避免表达式出现在查询中，而是先手工私下代数运算，转换为无表达式的查询语句。

## 索引选择性与前缀索引

既然索引可以加快查询速度，那么是不是只要是查询语句需要，就建上索引？答案是否定的。因为索引虽然加快了查询速度，但索引也是有代价的：索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，MySQL在运行时也要消耗资源维护索引，因此索引并不是越多越好。一般两种情况下不建议建索引。

第一种情况是表记录比较少，例如一两千条甚至只有几百条记录的表，没必要建索引，让查询做全表扫描就好了。至于多少条记录才算多，这个个人有个人的看法，我个人的经验是以2000作为分界线，记录数不超过 2000可以考虑不建索引，超过2000条可以酌情考虑索引。

另一种不建议建索引的情况是索引的选择性较低。所谓索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数（#T）的比值：

Index Selectivity = Cardinality / #T

显然选择性的取值范围为(0, 1]，选择性越高的索引价值越大，这是由B+Tree的性质决定的。例如，上文用到的employees.titles表，如果title字段经常被单独查询，是否需要建索引，我们看一下它的选择性：

1. SELECT count(DISTINCT(title))/count(\*) AS Selectivity FROM employees.titles;
2. +-------------+
3. | Selectivity |
4. +-------------+
5. | 0.0000 |
6. +-------------+

title的选择性不足0.0001（精确值为0.00001579），所以实在没有什么必要为其单独建索引。

有一种与索引选择性有关的索引优化策略叫做前缀索引，就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。下面以employees.employees表为例介绍前缀索引的选择和使用。

从图12可以看到employees表只有一个索引<emp\_no>，那么如果我们想按名字搜索一个人，就只能全表扫描了：

1. EXPLAIN SELECT \* FROM employees.employees WHERE first\_name='Eric' AND last\_name='Anido';
2. +----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+
3. | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |
4. +----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+
5. | 1 | SIMPLE | employees | ALL | NULL | NULL | NULL | NULL | 300024 | Using where |
6. +----+-------------+-----------+------+---------------+------+---------+------+--------+-------------+

如果频繁按名字搜索员工，这样显然效率很低，因此我们可以考虑建索引。有两种选择，建<first\_name>或<first\_name, last\_name>，看下两个索引的选择性：

1. SELECT count(DISTINCT(first\_name))/count(\*) AS Selectivity FROM employees.employees;
2. +-------------+
3. | Selectivity |
4. +-------------+
5. | 0.0042 |
6. +-------------+
7. SELECT count(DISTINCT(concat(first\_name, last\_name)))/count(\*) AS Selectivity FROM employees.employees;
8. +-------------+
9. | Selectivity |
10. +-------------+
11. | 0.9313 |
12. +-------------+

<first\_name>显然选择性太低，<first\_name, last\_name>选择性很好，但是first\_name和last\_name加起来长度为30，有没有兼顾长度和选择性的办法？可以考虑用first\_name和last\_name的前几个字符建立索引，例如<first\_name, left(last\_name, 3)>，看看其选择性：

1. SELECT count(DISTINCT(concat(first\_name, left(last\_name, 3))))/count(\*) AS Selectivity FROM employees.employees;
2. +-------------+
3. | Selectivity |
4. +-------------+
5. | 0.7879 |
6. +-------------+

选择性还不错，但离0.9313还是有点距离，那么把last\_name前缀加到4：

1. SELECT count(DISTINCT(concat(first\_name, left(last\_name, 4))))/count(\*) AS Selectivity FROM employees.employees;
2. +-------------+
3. | Selectivity |
4. +-------------+
5. | 0.9007 |
6. +-------------+

这时选择性已经很理想了，而这个索引的长度只有18，比<first\_name, last\_name>短了接近一半，我们把这个前缀索引 建上：

1. ALTER TABLE employees.employees
2. ADD INDEX `first\_name\_last\_name4` (first\_name, last\_name(4));

此时再执行一遍按名字查询，比较分析一下与建索引前的结果：

1. SHOW PROFILES;
2. +----------+------------+---------------------------------------------------------------------------------+
3. | Query\_ID | Duration | Query |
4. +----------+------------+---------------------------------------------------------------------------------+
5. | 87 | 0.11941700 | SELECT \* FROM employees.employees WHERE first\_name='Eric' AND last\_name='Anido' |
6. | 90 | 0.00092400 | SELECT \* FROM employees.employees WHERE first\_name='Eric' AND last\_name='Anido' |
7. +----------+------------+---------------------------------------------------------------------------------+

性能的提升是显著的，查询速度提高了120多倍。

前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于Covering index（即当索引本身包含查询所需全部数据时，不再访问数据文件本身）。

## InnoDB的主键选择与插入优化

在使用InnoDB存储引擎时，如果没有特别的需要，请永远使用一个与业务无关的自增字段作为主键。

经常看到有帖子或博客讨论主键选择问题，有人建议使用业务无关的自增主键，有人觉得没有必要，完全可以使用如学号或身份证号这种唯一字段作为主键。不论支持哪种论点，大多数论据都是业务层面的。如果从数据库索引优化角度看，使用InnoDB引擎而不使用自增主键绝对是一个糟糕的主意。

上文讨论过InnoDB的索引实现，InnoDB使用聚集索引，数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放，因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）。

如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。如下图所示：

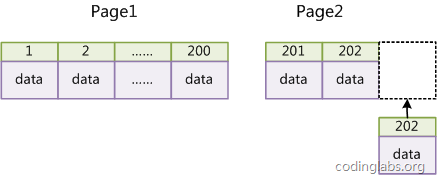


图13

这样就会形成一个紧凑的索引结构，近似顺序填满。由于每次插入时也不需要移动已有数据，因此效率很高，也不会增加很多开销在维护索引上。

如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置：

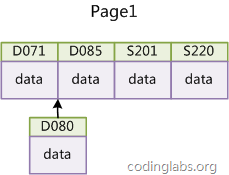


图14

此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

因此，只要可以，请尽量在InnoDB上采用自增字段做主键。