# 数据库(database)

## 数据库事务

### 四大特性(ACID)

#### 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

#### 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

#### 隔离性（Isolation）

事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

不考虑事务的隔离性，会发生的几种问题:

##### 脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

##### 不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

##### 幻读

幻读，因为事务1读取的数据状态并不能支持他的下一步的业务，见鬼了一样。

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

#### 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

　　例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

### 隔离级别

在MySQL数据库中，支持下面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

#### Serializable（串行化）

可避免脏读、不可重复读、幻读的发生。

#### Repeatable read（可重复读）

可避免脏读、不可重复读的发生。

#### Read committed（读已提交）

可避免脏读的发生。

#### Read uncommitted（读未提交）

最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。

## 数据库优化

### 百万级数据库优化方案

#### 避免全表扫描

对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

#### 避免null值判断

应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num = 0

#### 避免使用!=或<>操作符

应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

#### 避免使用or连接条件

应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or Name = 'admin'

可以这样查询：

select id from t where num = 10

union all

select id from t where Name = 'admin'

#### 慎用in和not in条件

in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

#### 慎用like条件

select id from t where name like ‘%abc%’

若要提高效率，可以考虑全文检索。

#### 避免在where子句中使用参数

如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num = @num

可以改为[强制使用索引](#_强制使用索引)：

select id from t with(index(索引名)) where num = @num

select /\*+ index(table IndexName)\*/\* from t where num = @num

#### 避免在 where 子句中对字段进行表达式操作

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2 = 100

应改为:

select id from t where num = 100\*2

#### 避免在where子句中对字段进行函数操作

应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3) = ’abc’ -–name以abc开头的id

select id from t where datediff(day,createdate,’2005-11-30′) = 0 -–‘2005-11-30’ --生成的id

应改为:

select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'

#### 避免在where 子句中的“=”左边进行函数、算术运算或其他表达式运算

不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

#### 复合索引必须使用该索引的第一个字段

在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

#### 先分页再join

对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

#### 索引并不是越多越好

索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

#### 应尽可能的避免更新[聚合](http://www.baidu.com/link?url=f3MEUUvzoVZ8wfzYsWu0b5p1u-9aoTrl4VsYr8Nfk_hZvTOTsaPCYmLgzGXSjskWfDGBBc_M11UZOSV7mkQN-K)(clustered)索引数据列

因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

#### 尽量使用数字型字段

若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连 接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

#### 尽可能的使用 varchar/nvarchar 代替 char/nchar

因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

#### 避免使用select \*

任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

#### 尽量使用表变量来代替临时表

如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

#### 避免频繁创建和删除临时表

避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

#### 新建临时表使用 select into 代替 create table

在新建临时表时，如果一次性插入数据量很大，那么可以使用select into代替create table，避免造成大量log，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

#### 临时表先 truncate table，然后 drop table

如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

#### 尽量避免使用游标

因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

#### 基于集的解决方案

使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

#### 游标并不是不可使用

与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

#### SET NOCOUNT ON和SET NOCOUNT OFF的使用

在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

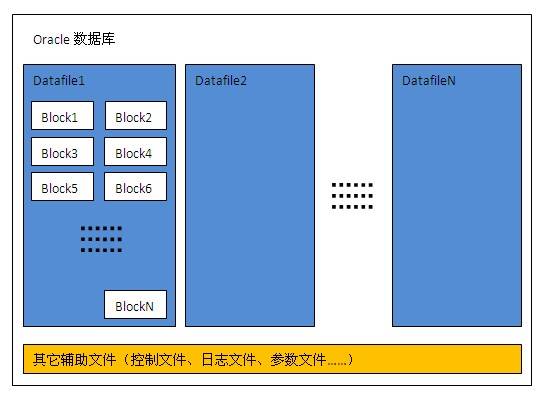
#### 尽量避免大事务操作

尽量避免大事务操作，提高系统并发能力。

## Oracle

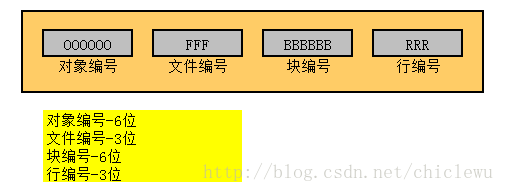
### 数据块(Block)

数据块是数据库中数据在磁盘中存储的最小单位，也是一次IO访问的最小单位，一个数据块通常可以存储多条记录，数据块大小是DBA在创建数据库或表空间时指定，可指定为2K、4K、8K、16K或32K字节。下图是一个[Oracle](http://lib.csdn.net/base/oracle)数据库典型的物理结构，一个数据库可以包括多个数据文件，一个数据文件内又包含多个数据块:



### ROWID

ROWID是每条记录在数据库中的唯一标识，通过ROWID可以直接定位记录到对应的文件号及数据块位置。ROWID内容包括对像号、文件号、数据块号、记录槽号，如下图所示：



### 索引

#### 强制使用索引

强制索引语法：SELECT /\*+index(table IndexName)\*/\* from table

如果查询语句中，表用到别名时，上边语句中的table一定要是别名，否则不走执行强制索引。

#### 索引使用条件

INDEX\_COLUMN = ?

INDEX\_COLUMN > ?

INDEX\_COLUMN >= ?

INDEX\_COLUMN < ?

INDEX\_COLUMN <= ?

INDEX\_COLUMN between ? and ?

INDEX\_COLUMN in (?,?,...,?)

INDEX\_COLUMN like ?||'%'（后导模糊查询）

T1. INDEX\_COLUMN=T2. COLUMN1（两个表通过索引字段关联）

#### 索引不使用条件

|  |  |
| --- | --- |
| 查询条件 | 不能使用索引原因 |
| INDEX\_COLUMN <> ?  INDEX\_COLUMN not in (?,?,...,?) | 不等于操作不能使用索引 |
| function(INDEX\_COLUMN) = ?  INDEX\_COLUMN + 1 = ?  INDEX\_COLUMN || 'a' = ? | 经过普通运算或函数运算后的索引字段不能使用索引 |
| INDEX\_COLUMN like '%'||?  INDEX\_COLUMN like '%'||?||'%' | 含前导模糊查询的Like语法不能使用索引 |
| INDEX\_COLUMN is null | B-TREE索引里不保存字段为NULL值记录，因此IS NULL不能使用索引 |
| NUMBER\_INDEX\_COLUMN='12345'  CHAR\_INDEX\_COLUMN=12345 | Oracle在做数值比较时需要将两边的数据转换成同一种数据类型，如果两边数据类型不同时会对字段值隐式转换，相当于加了一层函数处理，所以不能使用索引。 |
| a.INDEX\_COLUMN=a.COLUMN\_1 | 给索引查询的值应是已知数据，不能是未知字段值。 |

##经过函数运算的字段要使用索引可以使用**函数索引**，这种需求建议与DBA沟通。

有时候我们会使用多个字段的组合索引，如果查询条件中第一个字段不能使用索引，那整个查询也不能使用索引。

如：我们company表建了一个id+name的组合索引，以下SQL是不能使用索引的

select \* from company where name=?

Oracle9i后引入了一种index skip scan的索引方式来解决类似的问题，但是通过index skip scan提高性能的条件比较特殊，使用不好反而性能会更差。

#### 什么字段需要索引

主键及外键通常都要有索引。

字段出现在查询条件中，并且查询条件可以使用索引。

语句执行频率高，一天会有几千次以上。

通过字段条件可筛选的记录集很小，那数据筛选比例是多少才适合。

这个没有固定值，需要根据表数据量来评估，以下是经验公式，可用于快速评估：

小表(记录数小于10000行的表)：筛选比例<10%。

大表：(筛选返回记录数)<(表总记录数\*单条记录长度)/10000/16。

单条记录长度≈字段平均内容长度之和+字段数\*2。

以下是一些字段是否需要建B-TREE索引的经验分类：

|  |  |  |
| --- | --- | --- |
|  | 字段类型 | 常见字段名 |
| 需要建索引的字段 | 主键 | ID,PK |
| 外键 | PRODUCT\_ID,COMPANY\_ID,MEMBER\_ID,  ORDER\_ID,TRADE\_ID,PAY\_ID |
| 有对象或身份标识意义字段 | HASH\_CODE,USERNAME,IDCARD\_NO,  EMAIL,TEL\_NO,IM\_NO |
| 索引慎用字段,需要进行数据分布及使用场景详细评估 | 日期 | GMT\_CREATE,GMT\_MODIFIED |
| 年月 | YEAR,MONTH |
| 状态标志 | PRODUCT\_STATUS,ORDER\_STATUS,  IS\_DELETE,VIP\_FLAG |
| 类型 | ORDER\_TYPE,IMAGE\_TYPE,GENDER,  CURRENCY\_TYPE |
| 区域 | COUNTRY,PROVINCE,CITY |
| 操作人员 | CREATOR,AUDITOR |
| 数值 | LEVEL,AMOUNT,SCORE |
| 长字符 | ADDRESS,COMPANY\_NAME,  SUMMARY,SUBJECT |
| 不适合建索引的字段 | 描述备注 | DESCRIPTION,REMARK,MEMO,DETAIL |
| 大字段 | FILE\_CONTENT,EMAIL\_CONTENT |

#### B-TREE索引

B-TREE索引也称为平衡树索引(Balance Tree)，它是一种按字段排好序的树形目录结构，主要用于提升查询性能和唯一约束支持。B-TREE索引的内容包括**根节点**、**分支节点**、**叶子节点**。

**叶子节点内容**：索引字段内容+表记录ROWID。

**根节点，分支节点内容**：当一个数据块中不能放下所有索引字段数据时，就会形成树形的根节点或分支节点，根节点与分支节点保存了索引树的顺序及各层级间的引用关系。



##### 复合索引(组合索引)

一个索引由多个字段组成，称为组合索引。

对于复合索引，在Where限制条件中出现所有被索引的列时，优化器会选择走索引；

对于复合索引，在Where限制条件中出现联合索引中前导的列，即创建联合索引时前导的列时，优化器会选择走索引；

对于创建复合索引时，应考虑被索引字段的优先顺序，应将经常作为限制条件的字段放在首位；重复值少，即高基数(high-cardinaltiy)的列往前靠，而重复值多的字段往后靠；

##### 反向索引

##### 函数索引

#### 位图索引

#### 全文索引

### 执行计划

#### 执行计划常用列字段解释

基数（Rows） ##Oracle估计的当前操作的返回结果集行数。

字节（Bytes） ##执行该步骤后返回的字节数。

耗费（COST）、CPU耗费 ##Oracle估计的该步骤的执行成本，用于说明SQL执行的代价，理论上越小越好（该值可能与实际有出入）。

时间（Time） ##Oracle估计的当前操作所需的时间。

#### 表访问的几种方式

##### TABLE ACCESS FULL（全表扫描）

Oracle会读取表中所有的行，并检查每一行是否满足SQL语句中的 Where 限制条件；

全表扫描时可以使用多块读（即一次I/O读取多块数据块）操作，提升吞吐量；

使用建议：数据量太大的表不建议使用全表扫描，除非本身需要取出的数据较多，占到表数据总量的 5% ~ 10% 或以上。

##### **TABLE ACCESS BY ROWID（通过ROWID的表存取）**

ROWID是由Oracle自动加在表中每行最后的一列伪列，既然是伪列，就说明表中并不会物理存储ROWID的值。

你可以像使用其它列一样使用它，只是不能对该列的值进行增、删、改操作；

一旦一行数据插入后，则其对应的ROWID在该行的生命周期内是唯一的，即使发生行迁移，该行的ROWID值也不变。

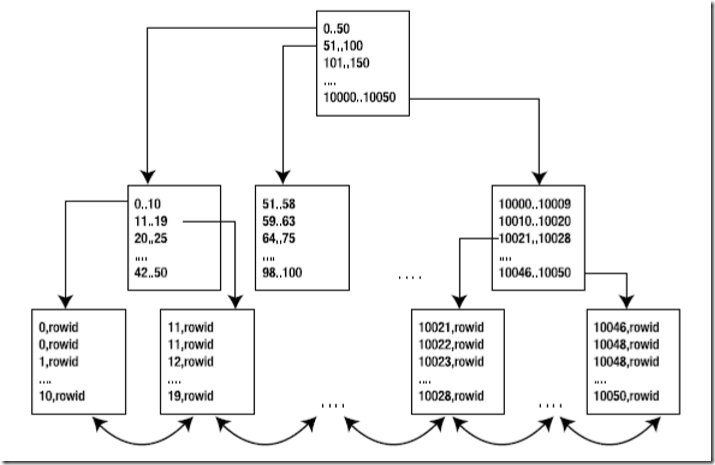
让我们再回到 TABLE ACCESS BY ROWID 来：

行的ROWID指出了该行所在的数据文件、数据块以及行在该块中的位置，所以通过ROWID可以快速定位到目标数据上，这也是Oracle中存取**单行**数据最快的方法。

##### **TABLE ACCESS BY INDEX SCAN（索引扫描）**

在索引块中，既存储每个索引的键值，也存储具有该键值的行的ROWID。

一个数字列上建索引后该索引可能的概念结构如下图：



索引扫描分为两步：

扫描索引得到对应的ROWID。

通过ROWID定位到具体的行读取数据。

#### 索引扫描类型

##### INDEX UNIQUE SCAN（索引唯一扫描）

针对唯一性索引（UNIQUE INDEX）的扫描，每次至多只返回一条记录；

表中某字段存在 UNIQUE、PRIMARY KEY 约束时，Oracle常实现唯一性扫描；

##### INDEX RANGE SCAN（索引范围扫描）

使用一个索引存取多行数据；

发生索引范围扫描的三种情况：

在唯一索引列上使用了范围操作符（如：>   <   <>   >=   <=   between）。

在组合索引上，只使用部分列进行查询（查询时必须包含前导列，否则会走全表扫描）。

对非唯一索引列上进行的任何查询。

##### INDEX FULL SCAN（索引全扫描）

进行全索引扫描时，查询出的数据都必须从索引中可以直接得到（注意全索引扫描只有在[CBO](#_CBO（Cost-Based_Optimization）_基于代价的优)模式下才有效）。

##### INDEX FAST FULL SCAN（索引快速扫描）

扫描索引中的所有数据块，与 [INDEX FULL SCAN](#_INDEX_FULL_SCAN（索引全扫描）) 类似，但是一个显著的区别是它不对查询出的数据进行排序（即数据不是以排序顺序被返回）。

##### INDEX SKIP SCAN（索引跳跃扫描）

Oracle 9i后提供，有时候[复合索引](#_组合索引(复合索引))的前导列（索引包含的第一列）没有在查询语句中出现，oralce也会使用该复合索引，这时候就使用的INDEX SKIP SCAN;

什么时候会触发 INDEX SKIP SCAN 呢？

前提条件：表有一个复合索引，且在查询时有除了前导列（索引中第一列）外的其他列作为条件，并且优化器模式为CBO时

当Oracle发现前导列的唯一值个数很少时，会将每个唯一值都作为常规扫描的入口，在此基础上做一次查找，最后合并这些查询；

例如：

假设表emp有ename（雇员名称）、job（职位名）、sex（性别）三个字段，并且建立了如 create index idx\_emp on emp (sex, ename, job) 的复合索引；

因为性别只有 '男' 和 '女' 两个值，所以为了提高索引的利用率，Oracle可将这个复合索引拆成 ('男', ename, job)，('女', ename, job) 这两个复合索引；

当查询 select \* from emp where job = 'Programmer' 时，该查询发出后：

Oracle先进入sex为'男'的入口，这时候使用到了 ('男', ename, job) 这条复合索引，查找 job = 'Programmer' 的条目；

再进入sex为'女'的入口，这时候使用到了 ('女', ename, job) 这条复合索引，查找 job = 'Programmer' 的条目；

最后合并查询到的来自两个入口的结果集。

### 优化器

#### RBO（Rule-Based Optimization） 基于规则的优化器

RBO有严格的使用规则，只要按照这套规则去写SQL语句，无论数据表中的内容怎样，也不会影响到你的执行计划；

换句话说，RBO对数据“不敏感”，它要求SQL编写人员必须要了解各项细则；

RBO一直沿用至ORACLE 9i，从ORACLE 10g开始，RBO已经彻底被抛弃。

#### CBO（Cost-Based Optimization） 基于代价的优化器

CBO是一种比RBO更加合理、可靠的优化器，在ORACLE 10g中完全取代RBO；

CBO通过计算各种可能的执行计划的“代价”，即COST，从中选用COST最低的执行方案作为实际运行方案；

它依赖数据库对象的统计信息，统计信息的准确与否会影响CBO做出最优的选择，也就是对数据“敏感”。

## MySQL

### 安装（5.7.13）

#### 系统约定

安装文件下载目录：**/data/software**

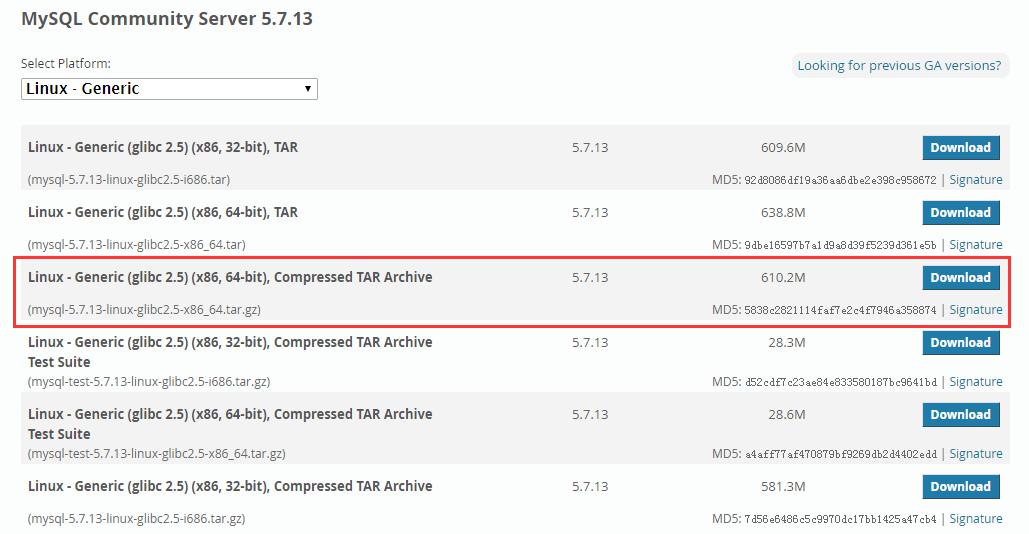
Mysql目录安装位置：**/usr/local/mysql**

数据库保存位置：**/data/mysql**

日志保存位置：**/data/log/mysql**

#### ****下载mysql****

在官网：<http://dev.mysql.com/downloads/mysql/> 中，选择以下版本的mysql下载：



执行如下命名：

#cd /data/software

#wget <http://dev.mysql.com/get/Downloads/MySQL-5.7/mysql-5.7.13-linux-glibc2.5-x86_64.tar.gz>

#### 解压压缩包到目标位置

# tar -xzvf /data/software/mysql-5.7.13-linux-glibc2.5-x86\_64.tar.gz

# mv mysql-5.7.13-linux-glibc2.5-x86\_64 /usr/local/mysql

#### 新建mysql用户、组及目录

# ---新建一个msyql组

# useradd -r -s /sbin/nologin -g mysql mysql -d /usr/local/mysql ---新建msyql用户禁止登录shell

#### 改变目录属有者

#cd /usr/local/mysql

#pwd

#chown –R mysql .

#chgrp –R mysql .

#chown -R mysql /data/mysql

#### 配置参数

#bin/mysqld --initialize --user=mysql --basedir=/usr/local/mysql --datadir=/data/mysql

**此处需要注意记录生成的临时密码**

#bin/mysql\_ssl\_rsa\_setup --datadir=/data/mysql

#### 修改系统配置文件

# cp my-default.cnf /etc/my.cnf

# cp mysql.server /etc/init.d/mysql

# vim /etc/init.d/mysql

修改以下内容：

basedir=/usr/local/mysql

datadir=/data/mysql

# vim /etc/my.cnf  
修改以下内容：

[client]

port = 3306

socket = /usr/local/mysql/mysql.sock

character-set-server = utf8

#### 启动mysql

#bin/mysqld\_safe --user=mysql &

bin/mysql --user=root –p

##输入第6步生成的临时密码

mysql> set password=password('123456');

mysql> grant all privileges on \*.\* to root@'%' identified by '123456';

mysql> flush privileges;

mysql> use mysql;

mysql> select host,user from user;

#### ****添加系统路径****

# vim /etc/profile

添加：

export PATH=/usr/local/mysql/bin:$PATH

### 启动/停止/重启Mysql

#### 查看版本

status;

select version();

#### 启动

##### 使用service启动

[root@localhost /]# service mysqld start (5.0版本是mysqld)

[root@szxdb etc]# service mysql start (5.5.7版本是mysql)

##### 使用mysqld脚本启动

/etc/inint.d/mysqld start

##### 使用 mysqld\_safe 启动

mysqld\_safe&

#### 停止

##### 使用 service 启动

service mysqld stop

##### 使用 mysqld 脚本启动

/etc/inint.d/mysqld stop

##### mysqladmin shutdown

#### 重启

##### 使用 service 启动

service mysqld restart

service mysql restart (5.5.7版本命令)

##### 使用 mysqld 脚本启动

/etc/init.d/mysqld restart

### my.cnf

#### 配置参数说明

# 以下选项会被MySQL客户端应用读取。

# 注意只有MySQL附带的客户端应用程序保证可以读取这段内容。

# 如果你想你自己的MySQL应用程序获取这些值。

# 需要在MySQL客户端库初始化的时候指定这些选项。

#

[client]

#password = [your\_password]

**port = 3306**

**socket = /usr/local/mysql/mysql.sock**

# \*\*\* 应用定制选项 \*\*\*

#

# MySQL 服务端

#

[mysqld]

# 一般配置选项

**port = 3306**

**socket = /usr/local/mysql/mysql.sock**

**basedir =/usr/local/mysql**

**datadir = /data/mysqldb**

# 修改back\_log参数值:由默认的50修改为500.（每个连接256kb, 占用：125M）

# back\_log=500

# 查看mysql 当前系统默认back\_log值，命令：

# show variables like 'back\_log';

# back\_log值指出在MySQL暂时停止回答新请求之前的短时间内多少个请求可以被存在堆栈中。

# 也就是说，如果MySql的连接数达到max\_connections时，新来的请求将会被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即back\_log，

# 如果等待连接的数量超过back\_log，将不被授予连接资源。将会报：

# unauthenticated user | xxx.xxx.xxx.xxx | NULL | Connect | NULL | login | NULL 的待连接进程时.

# back\_log值不能超过TCP/IP连接的侦听队列的大小。若超过则无效，查看当前系统的TCP/IP连接的侦听队列的大小命令：

# cat /proc/sys/net/ipv4/tcp\_max\_syn\_backlog，目前系统为1024。对于Linux系统推荐设置为大于512的整数。

修改系统内核参数，可以编辑/etc/sysctl.conf去调整它。如：net.ipv4.tcp\_max\_syn\_backlog = 2048，改完后执行sysctl -p 让修改立即生效。

**#back\_log = 300**

# 不在 TCP/IP 端口上进行监听.

# 如果所有的进程都是在同一台服务器连接到本地的 mysqld,

# 这样设置将是增强安全的方法

# 所有 mysqld 的连接都是通过 Unix Sockets 或者命名管道进行的.

# 注意在 Windows下如果没有打开命名管道选项而只是用此项

# (通过 “enable-named-pipe” 选项) 将会导致 MySQL 服务没有任何作用!

**#skip-networking**

# MySQL 服务所允许的同时会话数的上限

# 其中一个连接将被 SUPER 权限保留作为管理员登录.

**max\_connections = 3000**

# 每个客户端连接最大的错误允许数量,如果达到了此限制.

# 这个客户端将会被 MySQL 服务阻止直到执行了 “FLUSH HOSTS” 或者服务重启

# 非法的密码以及其他在链接时的错误会增加此值.

# 查看 “Aborted\_connects” 状态来获取全局计数器.

**max\_connect\_errors = 100**

table\_open\_cache指定表高速缓存的大小。每当MySQL访问一个表时，如果在表缓冲区中还有空间，该表就被打开并放入其中，这样可以更快地访问表内容。通过检查峰值时间的状态值Open\_tables和Opened\_tables，可以决定是否需要增加table\_open\_cache的值。如果你发现**open\_tables**等于**table\_open\_cache**，并且**opened\_tables**在不断增长，那么你就需要增加table\_open\_cache的值了(上述状态值可通过SHOW GLOBAL STATUS LIKE ‘Open%tables’获得)。

注意，不能盲目地把table\_open\_cache设置成很大的值，设置太大超过了shell的文件描述符(通过ulimit -n查看)，造成文件描述符不足，从而造成性能不稳定或者连接失败。

    MYSQL默认的table\_open\_cache为64，这个数值是偏小的，如果max\_connections较大，则容易引起性能问题。

    表现：数据库查询效率慢，show processlist 发现比较多的查询正在opening table。

    进一步确认，执行以下语句：

mysql> show global status like 'open%tables%';

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

| Variable\_name | Value   |

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

| Open\_tables   | 345     |

| Opened\_tables | 9734116 |

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

    Opened\_tables数值非常大，说明cache太小，导致要频繁地open table，可以查看下当前的table\_open\_cache设置：

mysql> show variables like '%table\_open\_cache%';

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

| Variable\_name    | Value |

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

| table\_open\_cache |     64|

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

     默认是64，一些资料推荐把这个数值设置为（max\_connections\* 查询同时用到的表数）。我实践中发现，一般设置为max\_connections就没问题了（如果还不够，可以继续加大，但不能设置大得离谱，可能会引发其他问题）。即时生效的设置：

mysql> set global table\_open\_cache=1024;

     设置后可以观察一下，如果opening table不再怎么出现，说明此修改是有效的，将其添加到mysql的配置文件，这样数据库重启后仍可保留此设置。

**table\_open\_cache = 4096**

# 允许外部文件级别的锁. 打开文件锁会对性能造成负面影响

# 所以只有在你在同样的文件上运行多个数据库实例时才使用此选项(注意仍会有其他约束!)

# 或者你在文件层面上使用了其他一些软件依赖来锁定 MyISAM 表

**#external-locking**

# 服务所能处理的请求包的最大大小以及服务所能处理的最大的请求大小(当与大的 BLOB 字段一起工作时相当必要)

# 每个连接独立的大小，大小动态增加

show variables like 'max\_allowed\_packet';

结果是1048576=1024K=1M

通过如下命令增大max\_allowed\_packet的值，解决值过小导致的问题

set global max\_allowed\_packet = 10\*1024\*1024

设置为10M，退出mysql，然后重新进入， 调用show variables like 'max\_allowed\_packet';查看是否修改成功  
注意事项：

1、 max\_allowed\_packet的值最大为1G，设置的值必须为1024的倍数  
2、设置完后，需要退出mysql，重新进入才能看到设置后的值

**max\_allowed\_packet = 128M**

# 在一个事务中 binlog 为了记录 SQL 状态所持有的 cache 大小

# 如果你经常使用大的,多声明的事务,你可以增加此值来获取更大的性能.

# 所有从事务来的状态都将被缓冲在 binlog 缓冲中然后在提交后一次性写入到 binlog 中

# 如果事务比此值大, 会使用磁盘上的临时文件来替代.

# 此缓冲在每个连接的事务第一次更新状态时被创建

binlog\_cache\_size = 4M

# 独立的内存表所允许的最大容量.

# 此选项为了防止意外创建一个超大的内存表导致永尽所有的内存资源.

max\_heap\_table\_size = 128M

# 随机读取数据缓冲区使用内存(read\_rnd\_buffer\_size)：和顺序读取相对应，

# 当 MySQL 进行非顺序读取（随机读取）数据块的时候，会利用>这个缓冲区暂存读取的数据

# 如根据索引信息读取表数据，根据排序后的结果集与表进行 Join 等等

# 总的来说，就是当数据块的读取需要满足>一定的顺序的情况下，MySQL 就需要产生随机读取，进而使用到 read\_rnd\_buffer\_size 参数所设置的内存缓冲区

read\_rnd\_buffer\_size = 16M

# 排序缓冲被用来处理类似 ORDER BY 以及 GROUP BY 队列所引起的排序

# 如果排序后的数据无法放入排序缓冲,一个用来替代的基于磁盘的合并分类会被使用

# 查看 “Sort\_merge\_passes” 状态变量.

# 在排序发生时由每个线程分配

sort\_buffer\_size = 16M

# 此缓冲被使用来优化全联合(FULL JOINS 不带索引的联合).

# 类似的联合在极大多数情况下有非常糟糕的性能表现,但是将此值设大能够减轻性能影响.

# 通过 “Select\_full\_join” 状态变量查看全联合的数量

# 当全联合发生时,在每个线程中分配

join\_buffer\_size = 16M

# 我们在 cache 中保留多少线程用于重用

# 当一个客户端断开连接后,如果 cache 中的线程还少于 thread\_cache\_size,则客户端线程被放入cache 中.

# 这可以在你需要大量新连接的时候极大的减少线程创建的开销

# (一般来说如果你有好的线程模型的话,这不会有明显的性能提升.)

thread\_cache\_size = 16

# 此允许应用程序给予线程系统一个提示在同一时间给予渴望被运行的线程的数量.

# 此值只对于支持 thread\_concurrency() 函数的系统有意义( 例如Sun Solaris).

# 你可可以尝试使用 [CPU数量]\*(2..4) 来作为 thread\_concurrency 的值

thread\_concurrency = 8

# 查询缓冲常被用来缓冲 SELECT 的结果并且在下一次同样查询的时候不再执行直接返回结果.

# 打开查询缓冲可以极大的提高服务器速度, 如果你有大量的相同的查询并且很少修改表.

# 查看 “Qcache\_lowmem\_prunes” 状态变量来检查是否当前值对于你的负载来说是否足够高.

# 注意: 在你表经常变化的情况下或者如果你的查询原文每次都不同,

# 查询缓冲也许引起性能下降而不是性能提升.

query\_cache\_size = 128M

# 只有小于此设定值的结果才会被缓冲

# 此设置用来保护查询缓冲,防止一个极大的结果集将其他所有的查询结果都覆盖.

query\_cache\_limit = 4M

# 被全文检索索引的最小的字长.

# 你也许希望减少它,如果你需要搜索更短字的时候.

# 注意在你修改此值之后,你需要重建你的 FULLTEXT 索引

ft\_min\_word\_len = 8

# 如果你的系统支持 memlock() 函数,你也许希望打开此选项用以让运行中的 mysql 在在内存高度紧张的时候,数据在内存中保持锁定并且防止可能被 swapping out

# 此选项对于性能有益

#memlock

# 当创建新表时作为默认使用的表类型,

# 如果在创建表示没有特别执行表类型,将会使用此值

#default\_table\_type = InnoDB

# 线程使用的堆大小. 此容量的内存在每次连接时被预留.

# MySQL 本身常不会需要超过 64K 的内存

# 如果你使用你自己的需要大量堆的 UDF 函数或者你的操作系统对于某些操作需要更多的堆,你也许需要将其设置的更高一点.

thread\_stack = 512K

# 设定默认的事务隔离级别.可用的级别如下:

# READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, SERIALIZABLE

transaction\_isolation = REPEATABLE-READ

# 内部(内存中)临时表的最大大小

# 如果一个表增长到比此值更大,将会自动转换为基于磁盘的表.

# 此限制是针对单个表的,而不是总和.

tmp\_table\_size = 128M

# 打开二进制日志功能.

# 在复制(replication)配置中,作为 MASTER 主服务器必须打开此项

# 如果你需要从你最后的备份中做基于时间点的恢复,你也同样需要二进制日志.

log-bin=/usr/local/mysql/logs/mysql-bin

# 如果你在使用链式从服务器结构的复制模式 (A->B->C),

# 你需要在服务器B上打开此项.

# 此选项打开在从线程上重做过的更新的日志, 并将其写入从服务器的二进制日志.

#log\_slave\_updates

# 打开全查询日志. 所有的由服务器接收到的查询 (甚至对于一个错误语法的查询)

# 都会被记录下来. 这对于调试非常有用, 在生产环境中常常关闭此项.

general\_log=ON

general\_log\_file=/usr/local/mysql/log/mysql.log

# 将警告打印输出到错误 log 文件. 如果你对于 MySQL 有任何问题

# 你应该打开警告 log 并且仔细审查错误日志,查出可能的原因.

#log\_warnings

# 记录慢速查询. 慢速查询是指消耗了比 “long\_query\_time” 定义的更多时间的查询.

# 如果 log\_long\_format 被打开,那些没有使用索引的查询也会被记录.

# 如果你经常增加新查询到已有的系统内的话. 一般来说这是一个好主意,

#log\_slow\_queries

# 所有的使用了比这个时间(以秒为单位)更多的查询会被认为是慢速查询.

# 不要在这里使用“1″, 否则会导致所有的查询,甚至非常快的查询页被记录下来(由于 MySQL 目前时间的精确度只能达到秒的级别).

long\_query\_time = 6

# 在慢速日志中记录更多的信息.

# 一般此项最好打开.

# 打开此项会记录使得那些没有使用索引的查询也被作为到慢速查询附加到慢速日志里

#log\_long\_format

# 此目录被MySQL用来保存临时文件.例如,

# 它被用来处理基于磁盘的大型排序,和内部排序一样.

# 以及简单的临时表.

# 如果你不创建非常大的临时文件,将其放置到 swapfs/tmpfs 文件系统上也许比较好

# 另一种选择是你也可以将其放置在独立的磁盘上.

# 你可以使用”;”来放置多个路径

# 他们会按照 roud-robin 方法被轮询使用.

#tmpdir = /tmp

# \*\*\* 主从复制相关的设置

# 唯一的服务辨识号,数值位于 1 到 2^32-1之间.

# 此值在master和slave上都需要设置.

# 如果 “master-host” 没有被设置,则默认为1, 但是如果忽略此选项,MySQL不会作为master生效.

server-id = 001

sync\_binlog=1

# 复制的Slave (去掉master段的注释来使其生效)

#

# 为了配置此主机作为复制的slave服务器,你可以选择两种方法:

#

# 1) 使用 CHANGE MASTER TO 命令 (在我们的手册中有完整描述) -

# 语法如下:

#

# CHANGE MASTER TO MASTER\_HOST=, MASTER\_PORT=,

# MASTER\_USER=, MASTER\_PASSWORD= ;

#

# 你需要替换掉 , , 等被尖括号包围的字段以及使用master的端口号替换 (默认3306).

#

# 例子:

#

# CHANGE MASTER TO MASTER\_HOST=’125.564.12.1′, MASTER\_PORT=3306,

# MASTER\_USER=’joe’, MASTER\_PASSWORD=’secret’;

#

# 或者

#

# 2) 设置以下的变量. 不论如何, 在你选择这种方法的情况下, 然后第一次启动复制(甚至不成功的情况下,

# 例如如果你输入错密码在master-password字段并且slave无法连接),

# slave会创建一个 master.info 文件,并且之后任何对于包含在此文件内的参数的变化都会被忽略

# 并且由 master.info 文件内的内容覆盖, 除非你关闭slave服务, 删除 master.info 并且重启slave 服务.

# 由于这个原因,你也许不想碰一下的配置(注释掉的) 并且使用 CHANGE MASTER TO (查看上面) 来代替

#

# 所需要的唯一id号位于 2 和 2^32 – 1之间

# (并且和master不同)

# 如果master-host被设置了.则默认值是2

# 但是如果省略,则不会生效

#server-id = 2

#

# 复制结构中的master – 必须

#master-host =

#

# 当连接到master上时slave所用来认证的用户名 – 必须

#master-user =

#

# 当连接到master上时slave所用来认证的密码 – 必须

#master-password =

#

# master监听的端口.

# 可选 – 默认是3306

#master-port =

# 使得slave只读.只有用户拥有SUPER权限和在上面的slave线程能够修改数据.

# 你可以使用此项去保证没有应用程序会意外的修改slave而不是master上的数据

#read\_only

#\*\*\* MyISAM 相关选项

# 关键词缓冲的大小, 一般用来缓冲 MyISAM 表的索引块.

# 不要将其设置大于你可用内存的30%,

# 因为一部分内存同样被OS用来缓冲行数据

# 甚至在你并不使用 MyISAM 表的情况下, 你也需要仍旧设置起 8-64M 内存由于它同样会被内部临时磁盘表使用.

key\_buffer\_size = 128M

# 用来做 MyISAM 表全表扫描的缓冲大小.

# 当全表扫描需要时,在对应线程中分配.

read\_buffer\_size = 8M

# 当在排序之后,从一个已经排序好的序列中读取行时,行数据将从这个缓冲中读取来防止磁盘寻道.

# 如果你增高此值,可以提高很多 ORDER BY 的性能.

# 当需要时由每个线程分配

read\_rnd\_buffer\_size = 64M

# MyISAM 使用特殊的类似树的 cache 来使得突发插入

# (这些插入是,INSERT … SELECT, INSERT … VALUES (…), (…), …, 以及 LOAD DATA INFILE) 更快.

# 此变量限制每个进程中缓冲树的字节数.

# 设置为 0 会关闭此优化.

# 为了最优化不要将此值设置大于 “key\_buffer\_size”.

# 当突发插入被检测到时此缓冲将被分配.

bulk\_insert\_buffer\_size = 256M

# 此缓冲当 MySQL 需要在 REPAIR, OPTIMIZE, ALTER 以及 LOAD DATA INFILE 到一个空表中引起重建索引时被分配.

# 这在每个线程中被分配.所以在设置大值时需要小心.

myisam\_sort\_buffer\_size = 256M

# MySQL 重建索引时所允许的最大临时文件的大小 (当 REPAIR, ALTER TABLE 或者 LOAD DATA INFILE).

# 如果文件大小比此值更大,索引会通过键值缓冲创建(更慢)

myisam\_max\_sort\_file\_size = 10G

# 如果被用来更快的索引创建索引所使用临时文件大于制定的值,那就使用键值缓冲方法.

# 这主要用来强制在大表中长字串键去使用慢速的键值缓冲方法来创建索引.

#myisam\_max\_extra\_sort\_file\_size = 10G

# 如果一个表拥有超过一个索引, MyISAM 可以通过并行排序使用超过一个线程去修复他们.

# 这对于拥有多个 CPU 以及大量内存情况的用户,是一个很好的选择.

myisam\_repair\_threads = 1

# 自动检查和修复没有适当关闭的 MyISAM 表.

myisam\_recover

# 默认关闭 Federated

skip-federated

# \*\*\* BDB 相关选项 \*\*\*

# 如果你运行的MySQL服务有BDB支持但是你不准备使用的时候使用此选项. 这会节省内存并且可能加速一些事.

#skip-bdb

# \*\*\* INNODB 相关选项 \*\*\*

# 如果你的 MySQL 服务包含 InnoDB 支持但是并不打算使用的话,

# 使用此选项会节省内存以及磁盘空间,并且加速某些部分

#skip-innodb

# 附加的内存池被 InnoDB 用来保存 metadata 信息(5.6中不再推荐使用)

# 如果 InnoDB 为此目的需要更多的内存,它会开始从 OS 这里申请内存.

# 由于这个操作在大多数现代操作系统上已经足够快, 你一般不需要修改此值.

# SHOW INNODB STATUS 命令会显示当先使用的数量.

innodb\_additional\_mem\_pool\_size = 64M

# InnoDB使用一个缓冲池来保存索引和原始数据, 不像 MyISAM.

# 这里你设置越大,这能保证你在大多数的读取操作时使用的是内存而不是硬盘,在存取表里面数据时所需要的磁盘 I/O 越少.

# 在一个独立使用的数据库服务器上,你可以设置这个变量到服务器物理内存大小的80%

# 不要设置过大,否则,由于物理内存的竞争可能导致操作系统的换页颠簸.

# 注意在32位系统上你每个进程可能被限制在 2-3.5G 用户层面内存限制,

# 所以不要设置的太高.

innodb\_buffer\_pool\_size = 6G

# InnoDB 将数据保存在一个或者多个数据文件中成为表空间.

# 如果你只有单个逻辑驱动保存你的数据,一个单个的自增文件就足够好了.

# 其他情况下.每个设备一个文件一般都是个好的选择.

# 你也可以配置 InnoDB 来使用裸盘分区 – 请参考手册来获取更多相关内容

innodb\_data\_file\_path = ibdata1:10M:autoextend

# 设置此选项如果你希望InnoDB表空间文件被保存在其他分区.

# 默认保存在MySQL的datadir中.

#innodb\_data\_home\_dir =

# 用来同步IO操作的IO线程的数量.

# 此值在Unix下被硬编码为8,但是在Windows磁盘I/O可能在一个大数值下表现的更好.

innodb\_file\_io\_threads = 8

# 如果你发现 InnoDB 表空间损坏, 设置此值为一个非零值可能帮助你导出你的表.

# 从1开始并且增加此值知道你能够成功的导出表.

#innodb\_force\_recovery=1

# 在 InnoDb 核心内的允许线程数量.

# 最优值依赖于应用程序,硬件以及操作系统的调度方式.

# 过高的值可能导致线程的互斥颠簸.

innodb\_thread\_concurrency = 16

# 如果设置为1 ,InnoDB 会在每次提交后刷新(fsync)事务日志到磁盘上,

# 这提供了完整的 ACID 行为.

# 如果你愿意对事务安全折衷, 并且你正在运行一个小的事物, 你可以设置此值到0或者2来减少由事务日志引起的磁盘I/O

# 0代表日志只大约每秒写入日志文件并且日志文件刷新到磁盘.

# 2代表日志写入日志文件在每次提交后,但是日志文件只有大约每秒才会刷新到磁盘上.

innodb\_flush\_log\_at\_trx\_commit = 2

#（说明：如果是游戏服务器，建议此值设置为2；如果是对数据安全要求极高的应用，建议设置为1；设置为0性能最高，但如果发生故障，数据可能会有丢失的危险！默认值1的意思是每一次事务提交或事务外的指令都需要把日志写入（flush）硬盘，这是很费时的。特别是使用电池供电缓存（Battery backed up cache）时。设成2对于很多运用，特别是从MyISAM表转过来的是可以的，它的意思是不写入硬盘而是写入系统缓存。日志仍然会每秒flush到硬盘，所以你一般不会丢失超过1-2秒的更新。设成0会更快一点，但安全方面比较差，即使MySQL挂了也可能会丢失事务的数据。而值2只会在整个操作系统挂了时才可能丢数据。）

# 加速 InnoDB 的关闭. 这会阻止 InnoDB 在关闭时做全清除以及插入缓冲合并.

# 这可能极大增加关机时间, 但是取而代之的是 InnoDB 可能在下次启动时做这些操作.

#innodb\_fast\_shutdown

# 用来缓冲日志数据的缓冲区的大小.

# 当此值快满时, InnoDB 将必须刷新数据到磁盘上.

# 由于基本上每秒都会刷新一次,所以没有必要将此值设置的太大(甚至对于长事务而言)

innodb\_log\_buffer\_size = 16M

# 在日志组中每个日志文件的大小.

# 你应该设置日志文件总合大小到你缓冲池大小的25%~100%

# 来避免在日志文件覆写上不必要的缓冲池刷新行为.

# 不论如何, 请注意一个大的日志文件大小会增加恢复进程所需要的时间.

innodb\_log\_file\_size = 512M

# 在日志组中的文件总数.

# 通常来说2~3是比较好的.

innodb\_log\_files\_in\_group = 3

# InnoDB 的日志文件所在位置. 默认是 MySQL 的 datadir.

# 你可以将其指定到一个独立的硬盘上或者一个RAID1卷上来提高其性能

#innodb\_log\_group\_home\_dir

# 在 InnoDB 缓冲池中最大允许的脏页面的比例.

# 如果达到限额, InnoDB 会开始刷新他们防止他们妨碍到干净数据页面.

# 这是一个软限制,不被保证绝对执行.

innodb\_max\_dirty\_pages\_pct = 90

# InnoDB 用来刷新日志的方法.

# 表空间总是使用双重写入刷新方法

# 默认值是 “fdatasync”, 另一个是 “O\_DSYNC”.

# 一般来说，如果你有硬件 RAID 控制器，并且其独立缓存采用 write-back 机制，并有着电池断电保护，那么应该设置配置为 O\_DIRECT

# 否则，大多数情况下应将其设为 fdatasync

#innodb\_flush\_method=fdatasync

# 在被回滚前,一个 InnoDB 的事务应该等待一个锁被批准多久.

# InnoDB 在其拥有的锁表中自动检测事务死锁并且回滚事务.

# 如果你使用 LOCK TABLES 指令, 或者在同样事务中使用除了 InnoDB 以外的其他事务安全的存储引擎

# 那么一个死锁可能发生而 InnoDB 无法注意到.

# 这种情况下这个 timeout 值对于解决这种问题就非常有帮助.

innodb\_lock\_wait\_timeout = 120

# 这项设置告知InnoDB是否需要将所有表的数据和索引存放在共享表空间里（innodb\_file\_per\_table = OFF） 或者为每张表的数据单独放在一个.ibd文件（innodb\_file\_per\_table = ON）

# 每张表一个文件允许你在drop、truncate或者rebuild表时回收磁盘空间

# 这对于一些高级特性也是有必要的，比如数据压缩,但是它不会带来任何性能收益

innodb\_file\_per\_table = on

[mysqldump]

# 不要在将内存中的整个结果写入磁盘之前缓存. 在导出非常巨大的表时需要此项

quick

max\_allowed\_packet = 32M

[mysql]

no-auto-rehash

# 仅仅允许使用键值的 UPDATEs 和 DELETEs .

#safe-updates

[myisamchk]

key\_buffer = 16M

sort\_buffer\_size = 16M

read\_buffer = 8M

write\_buffer = 8M

[mysqlhotcopy]

interactive-timeout

[mysqld\_safe]

# 增加每个进程的可打开文件数量.

# 警告: 确认你已经将全系统限制设定的足够高!

# 打开大量表需要将此值设大

### 数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 大小 | 范围(有符号) | 范围(无符号) |
| TINYINT | 1字节 | (-128,127) | （0，255） |
| SMALLINT | 2字节 | (-32768,32767) | （0，65535） |
| MEDIUMINT | 3字节 | (-8388608, 8388607) | （0，16777215） |
| INT | 4字节 | (-2147483648, 2147483647) | （0，4294967295） |
| BIGINT | 8字节 | (-9223372036854775808, 9223372036854775807) | （0，18446744073709551616） |
|  |  |  |  |
| float | 4 | 单精度浮点型 |  |
| double | 8 | 双精度浮点型 |  |
|  |  |  |  |
| date | 3 | yyyy-MM-dd | 存储日期值 |
| time | 3 | HH:mm:ss | 存储时分秒 |
| year | 1 | yyyy | 存储年 |
| datetime | 8 | yyyy-MM-dd HH:mm:ss | 存储日期+时间 |
| timestamp | 4 | yyyy-MM-dd HH:mm:ss | 存储日期+时间，可作时间戳 |
|  |  |  |  |
| char |  | 0-255字节 | 定长字符串 |
| varchar |  | 0-65535字节 | 变长字符串 |
| tinyblob |  | 0-255字节 | 不超过255个字符的二进制字符串 |
| tinytext |  | 0-255字节 | 短文本字符串 |
| blob |  | 0-65535字节 | 二进制形式的长文本数据 |
| text |  | 0-65535字节 | 长文本数据 |
| mediumblob |  | 0-16777215字节 | 二进制形式的中等长度文本数据 |
| mediumtext |  | 0-16777215字节 | 中等长度文本数据 |
| longblob |  | 0-4294967295字节 | 二进制形式的极大文本数据 |
| longtext |  | 0-4294967295字节 | 极大文本数据 |

### 命令

#### 基础命令

##### mysql连接

mysql -h主机地址 -u用户名 -p用户密码

##### 数据库操作

show databases\G;##显示所有数据库

create database name; ##创建数据库

use databasename; ##选择数据库

drop database name ##直接删除数据库，不提醒

##### 表操作

show tables; ##显示表

describe tablename; ##表的详细描述

##### 修改密码

set password=password('123456');

##### 查看数据库用户

mysql> use mysql;  
mysql> select host,user from user\G;

##### 新增用户(授权)

grant select on [数据库](http://www.cr173.com/k/sql/).\* to 用户名@登录主机 identified by \"密码\"

增加一个用户[test](http://www.cr173.com/test.html)1密码为abc，让他可以在任何主机上登录，并对所有数据库有查询、插入、修改、删除的权限。

首先用以root用户连入MySQL，然后键入以下命令：

grant select,insert,update,delete on \*.\* to test2@localhost identified by "abc";

如果你不想test2有密码，可以再打一个命令将密码消掉。

grant select,insert,update,delete on mydb.\* to test2@localhost identified by "";

##### 查看用户权限

mysql>**show grants for 用户;**

##### 删除用户

mysql>**delete from user where user='test' and host='localhost';**

 　　mysql>**flush privileges;**

 　　mysql>**drop database testDB;** //删除用户的数据库

删除账户及权限：>**drop user 用户名@'%';**

　　　　　　　　>**drop user 用户名@localhost;**

### 数据库引擎

#### MyISAM

MySql的默认引擎，**不支持事务，行级锁和外键**。

当执行Insert插入和Update**更新语句时**，即执行写操作的时候需要**锁定整个表**，**导致效率降低**。

不过和Innodb不同的是，MyIASM引擎是**保存了表的行数**，于是当进行Select count(\*) from table语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的**读操作远远多于写操作**时，并且**不需要事务的支持**，可以**选MyIASM作为数据库引擎**。

##### 数据库表文件

**.MYD**文件：即MY Data，表数据文件

**.MYI**文件：即MY Index，索引文件

**.log**文件：日志文件

#### InnoDB

Innodb引擎提供了**对数据库ACID事务的支持**。

**提供行级锁和外键的约束**。它的设计的目标就是处理大数据容量的数据库系统。它本身实际上是基于Mysql后台的完整的系统。Mysql运行的时候，Innodb会在内存中**建立缓冲池，用于缓冲数据和索引**。但是，该引擎是**不支持全文搜索**的。同时，启动也比较的慢，它**不保存表的行数**。当进行Select count(\*) from table指令的时候，需要进行扫描全表。所以当需要使用数据库的事务时，该引擎就是首选。由于**锁的粒度小**，**写操作不会锁定全表**。所以在并发度较高的场景下使用会提升效率的。

##### 数据库表文件

InnoDB采用表空间（tablespace）来管理数据，存储表数据和索引，

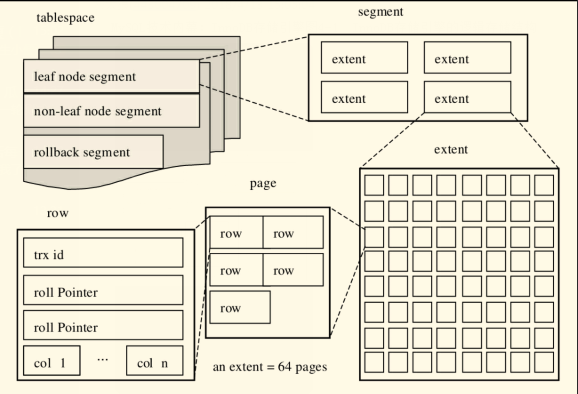
InnoDB数据库文件（即InnoDB文件集，ib-file set）：

**ibdata1**、**ibdata2**等：系统表空间文件，存储InnoDB系统信息和用户数据库表数据和索引，所有表共用

**.ibd**文件：单表表空间文件，**每个表使用一个表空间文件**（file per table），存放用户数据库表数据和索引

日志文件： ib\_logfile1、ib\_logfile2

##### 逻辑存储结构



1. **表空间（tablespace）**

InnoDB存储引擎的逻辑存储结构和Oracle大致相同，所有数据都被逻辑地存放在一个空间中，我们称之为**表空间（tablespace）**，

表空间又由**段（segment）**、**区（extent）**、**页（page）**组成。页在一些文档中有时也称为**块（block）。**

默认情况下InnoDB存储引擎有一个共享表空间ibdata1，即所有数据都放在这个表空间内。如果我们启用了参数**innodb\_file\_per\_table**，则每张表内的数据可以单独放到一个表空间内。

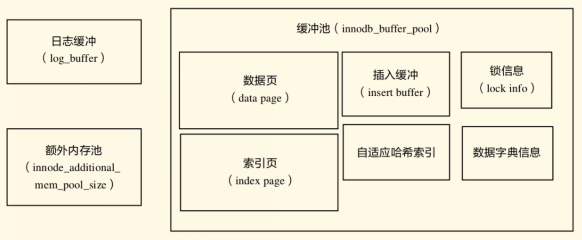
**mysql>show variables like 'innodb\_file\_table'\G;**

**mysql>system ls -lh /usr/local/mysql/data/ib\***

对于启用了innodb\_file\_per\_table的参数选项，需要注意的是，每张表的表空间内存放的只是数据、索引和插入缓冲，其他类的数据，如撤销（Undo）信息、系统事务信息、二次写缓冲（double write buffer）等还是存放在原来的共享表空间内。这也就说明了另一个问题：即使在启用了参数innodb\_file\_per\_table之后，共享表空间还是会不断地增加其大小。

1. **段（segment）**
2. **区（extent）**
3. **页（page）**

##### 内存



1. **缓冲池(innodb\_buffer\_pool\_size)**

缓冲池是占最大块内存的部分，用来存放各种数据的缓存。因为InnoDB的存储引擎的工作方式总是将数据库文件按页（每页16K）读取到缓冲池，然后按最近最少使用（LRU）的算法来保留在缓冲池中的缓存数据。如果数据库文件需要修改，总是首先修改在缓存池中的页（发生修改后，该页即为脏页），然后再按照一定的频率将缓冲池的脏页刷新（flush）到文件。可以通过命令SHOW ENGINE INNODB STATUS来查看innodb\_buffer\_pool的具体使用情况。

**mysql > show engine innodb status\G;**

**……**

**Per second averages calculated from the last 57 seconds ##信息是过去57秒内的数据库状态。**

……

----------------------

**BUFFER POOL AND MEMORY**

----------------------

Total large memory allocated 137428992

Dictionary memory allocated 102398

Buffer pool size 8191 **##一共有8191个缓冲帧(buffer frame)，每个缓冲帧为16K**

Free buffers 7929 **##当前空闲的缓冲帧**

Database pages 262 **##已经使用的缓冲帧**

Old database pages 0

Modified db pages 0 **##脏页的数量**

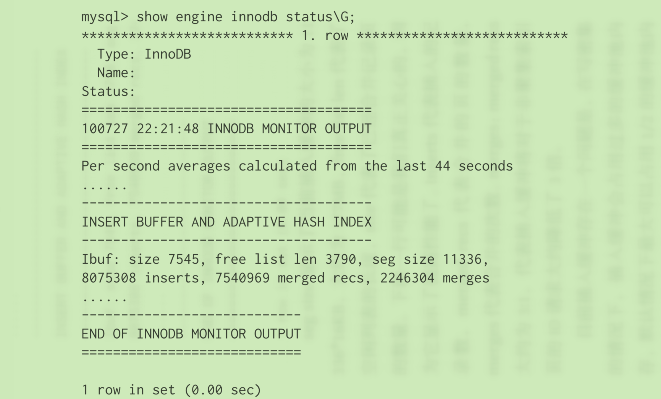
……

1. **重做日志缓冲池(innodb\_log\_buffer\_size)**
2. **额外内存池(innodb\_additional\_mem\_pool\_size)**

##### 插入缓冲（insert buffer)

插入缓冲,并不是缓存的一部分,而是物理页,对于非聚集索引的插入或更新操作,不是每一次直接插入索引页.而是先判断插入的非聚集索引页是否在缓冲池中.如果在,则直接插入,如果不再,则先放入一个插入缓冲区中.然后再以一定的频率执行插入缓冲和非聚集索引页子节点的合并操作。

使用条件:**非聚集索引**,**非唯一**。

****

**seg size**显示了当前插入缓冲的大小为2\*16KB，**free list len**代表了空闲列表的长度，**size**代表了已经合并记录页的数量。下面一行可能是我们真正关心的，因为它显示了提高性能了。**Inserts**代表插入的记录数，**merged recs**代表合并的页的数量，**merges**代表合并的次数。merges：merged recs大约为3:1，代表插入缓冲将对于非聚集索引页的IO请求大约降低了3倍。

目前插入缓冲存在一个问题是，在写密集的情况下，插入缓冲会占用过多的缓冲池内存，默认情况下最大可以占用1/2的缓冲池内存。

修改IBUF\_POOL\_SIZE\_PER\_MAX\_SIZE就可以对插入缓冲的大小进行控制，例如，将IBUF\_POOL\_SIZE\_PER\_MAX\_SIZE改为3，则最大只能使用1/3的缓冲池内存。

##### 二次写(double write)

##### 自适应哈希索引(ahi)

##### 预读(read ahead)

#### MyISAM与InnoDB的区别

|  |  |  |
| --- | --- | --- |
|  | MyISAM | InnoDB |
| 事务 | 不支持 | **支持** |
| 行级锁 | 不支持 | **支持** |
| MVCC | 不支持 | **支持** |
| 外键 | 不支持 | **支持** |
| 多种行格式 | 不支持 | **支持** |
| 全文索引 | **支持** | 不支持 |
| 拷贝表文件到另一台机器 | **支持** | 不支持 |
| 表类型 | 堆表 | 索引组织表 |

### 索引

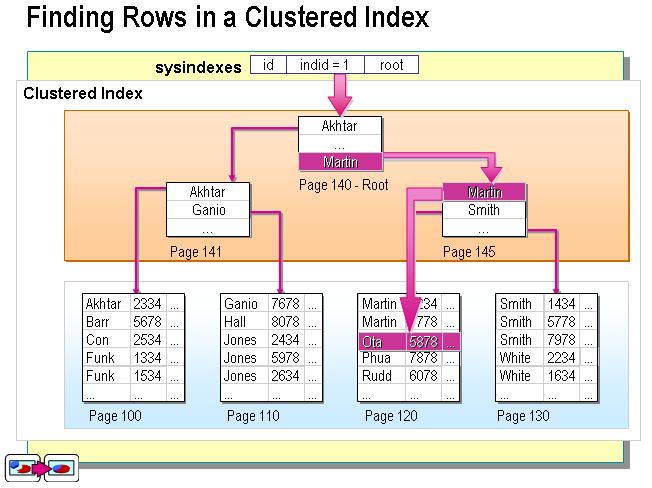
#### 聚集索引(聚簇索引)

##### 聚集索引

该索引中键值的逻辑顺序决定了表中相应行的物理顺序。**聚集索引**的叶节点就是数据节点。一个表只能建一个**聚集索引。**

innodb的聚集索引实际上是在同一个结构中保存了btree索引和数据行。

Innodb的存储索引是基于B+tree，理所当然，聚集索引也是基于B+tree。与非聚集索引的区别则是，聚集索引既存储了索引，也存储了行值。当一个表有一个聚集索引，它的数据是存储在索引的叶子页（leaf pages）。因此innodb也能理解为基于索引的表。



确定物理顺序

**聚集索引**确定表中数据的物理顺序。**聚集索引**类似于电话簿，后者按姓氏排列数据。由于**聚集索引**规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。

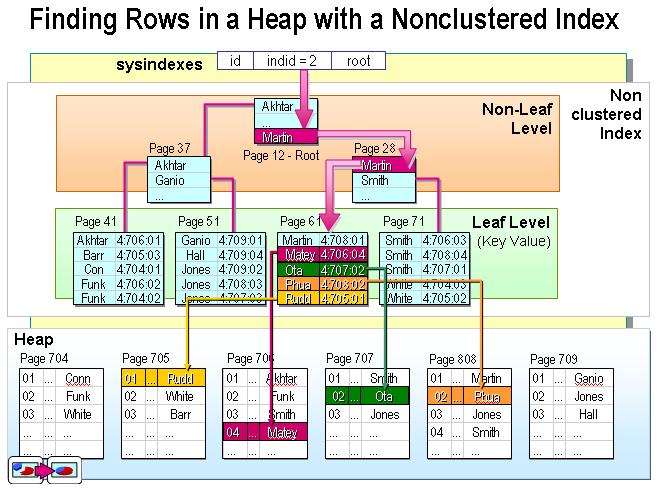
搜索范围值特别有效

**聚集索引**对于那些经常要搜索范围值的列特别有效。使用**聚集索引**找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻。例如，如果应用程序执行 的一个查询经常检索某一日期范围内的记录，则使用聚集索引可以迅速找到包含开始日期的行，然后检索表中所有相邻的行，直到到达结束日期。这样有助于提高此 类查询的性能。同样，如果对从表中检索的数据进行排序时经常要用到某一列，则可以将该表在该列上聚集（物理排序），避免每次查询该列时都进行排序，从而节 省成本。

当索引值唯一时，使用聚集索引查找特定的行也很有效率。例如，使用唯一雇员 ID 列 emp\_id 查找特定雇员的最快速的方法，是在 emp\_id 列上创建聚集索引或 PRIMARY KEY 约束。

##### 非聚集索引

该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。



##### Innodb如何选择一个聚集索引

对于Innodb，主键毫无疑问是一个聚集索引。但是当一个表没有主键，或者没有一个索引，Innodb会如何处理呢。请看如下规则

1. 如果一个主键被定义了，那么这个主键就是作为**聚集索引。**
2. 如果没有主键被定义，那么该表的第一个**唯一非空索引**被作为**聚集索引。**
3. 如果没有主键也没有合适的唯一索引，那么innodb内部会生成**一个隐藏的主键**作为聚集索引，这个隐藏的主键是一个6个字节的列，改列的值会随着数据的插入自增。

还有一个需要注意的是：

**次级索引**的叶子节点并不存储行数据的物理地址。而是存储的该行的主键值。

所以：一**次级索引**包含了两次查找。一次是查找**次级索引**自身。然后查找主键（聚集索引）。

##### 何时使用聚集索引或非聚集索引

下面的表总结了何时使用聚集索引或非聚集索引（很重要）：

|  |  |  |
| --- | --- | --- |
| 动作描述 | 使用聚集索引 | 使用非聚集索引 |
| 列经常被分组排序 | 应 | 应 |
| 返回某范围内的数据 | 应 | 不应 |
| 一个或极少不同值 | 不应 | 不应 |
| 小数目的不同值（distinct 小） | 应 | 不应 |
| 大数目的不同值（distinct 大） | 不应 | 应 |
| 频繁更新的列 | 不应 | 应 |
| 外键列 | 应 | 应 |
| 主键列 | 应 | 应 |
| 频繁修改索引列 | 不应 | 应 |

##### 聚集索引的优点

1. 区间查找非常快速。
2. 排序非常快。

如果一个表的某个字段经常被用来排序，如果该字段在聚集索引列，则排序的时间就是查询的时间。同时，因为B-Tree同时保存了索引和数据值。因此通过聚集索引的检索会快于一个非聚集索引的检索。

##### 聚集索引的缺点

1. 如果一个很大的聚集索引被定义。那么所有的次级索引也会显而易见的大，因为次级索引存储了聚集索引的key。因为数据的存储方式，次级索引的查找需要两次。
2. 聚集索引的列的值被更改后，聚集索引会对数据重排，因此主键被更改的开销是昂贵的。
3. 插入可能比较慢，如果数据不是按照主键的顺序插入。因此我们判断插入的数据的效率决定于插入的顺序。在innodb的表中，插入行按照主键的顺序插入式是最快的方式。

#### 次级索引（secondary index）

除了聚簇索引之外的其他索引类型都属于次级索引。

在innodb，每个次级索引都存储的主键key。因为innodb的数据结构决定了，次级索引存储的是主键的值。次级索引的叶子节点并不存储行数据的物理地址。而是存储的该行的主键值。因此，换句话说，主键扮演了行数据的指针。

一次级索引包含了两次查找。一次是查找次级索引自身。然后查找主键（聚集索引）。

#### 索引使用的误区

##### 主键就是聚集索引(误区)

这种想法笔者认为是极端错误的，是对聚集索引的一种浪费。虽然SQL SERVER默认是在主键上建立聚集索引的。

通常，我们会在每个表中都建立一个ID列，以区分每条数据，并且这个ID列是自动增大的，步长一般为1。我们的这个办公自动化的实例中的列Gid就是如此。此时，如果我们将这个列设为主键，SQL SERVER会将此列默认为聚集索引。这样做有好处，就是可以让您的数据在数据库中按照ID进行物理排序，但笔者认为这样做意义不大。

显而易见，聚集索引的优势是很明显的，而每个表中只能有一个聚集索引的规则，这使得聚集索引变得更加珍贵。

从我们前面谈到的聚集索引的定义我们可以看出，使用聚集索引的最大好处就是能够根据查询要求，迅速缩小查询范围，避免全表扫描。在实际应用中，因为 ID号是自动生成的，我们并不知道每条记录的ID号，所以我们很难在实践中用ID号来进行查询。这就使让ID号这个主键作为聚集索引成为一种资源浪费。其次，让每个ID号都不同的字段作为聚集索引也不符合“大数目的不同值情况下不应建立聚合索引”规则；当然，这种情况只是针对用户经常修改记录内容，特别是索引项的时候会负作用，但对于查询速度并没有影响。

在办公自动化系统中，无论是系统首页显示的需要用户签收的文件、会议还是用户进行文件查询等任何情况下进行数据查询都离不开字段的是“日期”还有用户本身的“用户名”。

通常，办公自动化的首页会显示每个用户尚未签收的文件或会议。虽然我们的where语句可以仅仅限制当前用户尚未签收的情况，但如果您的系统已建立了很长时间，并且数据量很大，那么，每次每个用户打开首页的时候都进行一次全表扫描，这样做意义是不大的，绝大多数的用户1个月前的文件都已经浏览过了，这样做只能徒增数据库的开销而已。事实上，我们完全可以让用户打开系统首页时，数据库仅仅查询这个用户近3个月来未阅览的文件，通过“日期”这个字段来限制表扫描，提高查询速度。如果您的办公自动化系统已经建立的2年，那么您的首页显示速度理论上将是原来速度8倍，甚至更快。

##### 只要建立索引就能显著提高查询速度(误区)

  （1）仅在主键上建立聚集索引，并且不划分时间段：

Select gid,fariqi,neibuyonghu,title from tgongwen

用时：128470毫秒（即：128秒）

（2）在主键上建立聚集索引，在fariq上建立非聚集索引：

select gid,fariqi,neibuyonghu,title from Tgongwen

where fariqi> dateadd(day,-90,getdate())

用时：53763毫秒（54秒）

（3）将聚合索引建立在日期列（fariqi）上：

select gid,fariqi,neibuyonghu,title from Tgongwen

where fariqi> dateadd(day,-90,getdate())

用时：2423毫秒（2秒）

事实上，我们可以发现上面的例子中，第2、3条语句完全相同，且建立索引的字段也相同；不同的仅是前者在fariqi字段上建立的是非聚合索引，后者在此字段上建立的是聚合索引，但查询速度却有着天壤之别。所以，并非是在任何字段上简单地建立索引就能提高查询速度。

从建表的语句中，我们可以看到这个有着1000万数据的表中fariqi字段有5003个不同记录。在此字段上建立聚合索引是再合适不过了。在现实中，我们每天都会发几个文件，这几个文件的发文日期就相同，这完全符合建立聚集索引要求的：“既不能绝大多数都相同，又不能只有极少数相同”的规则。由此看来，我们建立“适当”的聚合索引对于我们提高查询速度是非常重要的。

##### 把所有需要提高查询速度的字段都加进聚集索引，以提高查询速度

如果仅用聚集索引的起始列作为查询条件和同时用到复合聚集索引的全部列的查询速度是几乎一样的，甚至比用上全部的复合索引列还要略快（在查询结果集数目一样的情况下）；而如果仅用复合聚集索引的非起始列作为查询条件的话。同时，请记住：无论您是否经常使用聚合索引的其他列，但其前导列一定要是使用最频繁的列。

### 数据库优化

#### 开启慢查询日志

查看所有日志状态： show variables like '%quer%';

查看慢查询状态：show variables like 'show%'

vim /etc/my.cnf

[mysqld]

slow\_query\_log = on ##开启慢查询功能

slow\_query\_log\_file = /usr/local/mysql/data/slow-query.log ##慢查询日志存放路径与名称

long\_query\_time = 5 ##查询时间超过5s的查询语句

log\_queries\_not\_using\_indexes = on ##列出没有使用索引的查询语句

#### ****explain（执行计划）****

explain返回各列的含义

**extra**列需要注意的返回值

**Using filesort**:看到这个的时候，查询就需要优化了。MYSQL需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行

**Using temporary**看到这个的时候，查询需要优化了。这里，MYSQL需要创建一个临时表来存储结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上

**table**:显示这一行的数据是关于哪张表的。

**type**:这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为const、eq\_reg、ref、range、index 和ALL。

**possible\_keys**:显示可能应用在这张表中的索引。如果为空，没有可能的索引。

**key**:实际使用的索引。如果为NULL,则没有使用索引。

**keyjen**: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。

**ref**:显示索引的哪一列被使用了，如果可能的话，是一个常数。

**rows**: MYSQL认为必须检查的用来返回请求数据的行数。

#### sql优化

##### ****子查询的优化****

通常情况下把子查询优化为join查询，但在优化的时候需要注意关联建是否有一对多的关系，要特别注意重复数据