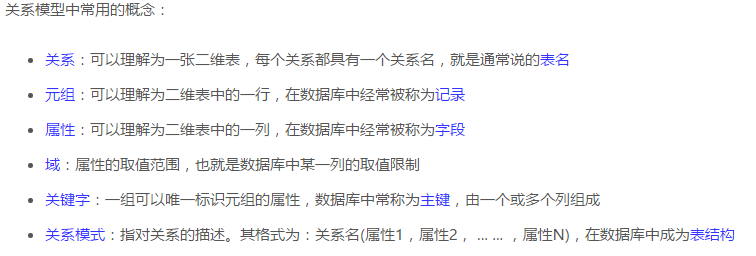
# 数据库

## 关系型

关系型数据库是指采用了关系模型来组织数据的数据库。

关系模型指的是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。



**瓶颈：**

* 高并发读写需求

网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈

* 海量数据的高效率读写

网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的

* 高扩展性和可用性

在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移。

对网站来说，关系型数据库的很多特性不再需要了：

* **事务一致性**

关系型数据库在对事物一致性的维护中有很大的开销，而现在很多web2.0系统对事物的读写一致性都不高

* **读写实时性**

对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比如发一条消息之后，过几秒乃至十几秒之后才看到这条动态是完全可以接受的

* 复杂SQL，特别是**多表关联查询**

任何**大数据**量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询，特别是SNS类型的网站，从需求以及产品阶级角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能极大的弱化了

在关系型数据库中，导致性能欠佳的最主要原因是多表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询。为了保证数据库的ACID特性，我们必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的**数据结构**。每个元组字段的组成都是一样，即使不是每个元组都需要所有的字段，但数据库会为每个元组分配所有的字段，这样的结构可以便于标语表之间进行链接等操作，但从另一个角度来说它也是关系型数据库性能瓶颈的一个因素。

* **高并发读写需求**

**网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈**

* **海量数据的高效率读写**

**网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的**

* **高扩展性和可用性**

**在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移。**

**对网站来说，关系型数据库的很多特性不再需要了：**

* **事务一致性**

**关系型数据库在对事物一致性的维护中有很大的开销，而现在很多web2.0系统对事物的读写一致性都不高**

* **读写实时性**

**对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比如发一条消息之后，过几秒乃至十几秒之后才看到这条动态是完全可以接受的**

* **复杂SQL，特别是多表关联查询**

**任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询，特别是SNS类型的网站，从需求以及产品阶级角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能极大的弱化了**

**在关系型数据库中，导致性能欠佳的最主要原因是多表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询。为了保证数据库的ACID特性，我们必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的数据结构。每个元组字段的组成都是一样，即使不是每个元组都需要所有的字段，但数据库会为每个元组分配所有的字段，这样的结构可以便于标语表之间进行链接等操作，但从另一个角度来说它也是关系型数据库性能瓶颈的一个因素。**

## 非关系型

用于指代那些非关系型的，分布式的，且一般不保证遵循ACID原则的数据存储系统。

注：数据库事务必须具备ACID特性，ACID是Atomic原子性，Consistency一致性，Isolation隔离性，Durability持久性。

非关系型数据库提出另一种理念，例如，以键值对存储，且结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。使用这种方式，用户可以根据需要去添加自己需要的字段，这样，为了获取用户的不同信息，不需要像关系型数据库中，要对多表进行关联查询。仅需要根据id取出相应的value就可以完成查询。但非关系型数据库由于很少的约束，他也不能够提供像SQL所提供的where这种对于字段属性值情况的查询。并且难以体现设计的完整性。他只适合存储一些较为简单的数据，对于需要进行较复杂查询的数据，SQL数据库显的更为合适。

* 面向高性能并发读写的key-value数据库：

key-value数据库的主要特点即使具有极高的并发读写性能，[**Redis**](http://lib.csdn.net/base/redis),Tokyo Cabinet,Flare就是这类的代表

* 面向海量数据访问的面向文档数据库：

这类数据库的特点是，可以在海量的数据中快速的查询数据，典型代表为[**MongoDB**](http://lib.csdn.net/base/mongodb)以及CouchDB

* 面向可扩展性的分布式数据库：

这类数据库想解决的问题就是传统数据库存在可扩展性上的缺陷，这类数据库可以适应数据量的增加以及数据结构的变化

## 关系型V非关系型

关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有ACID的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。但是，在网页应用中，尤其是SNS应用中，一致性却不是显得那么重要，用户A看到的内容和用户B看到同一用户C内容更新不一致是可以容忍的，或者说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。

相反地，**关系型数据库为了维护一致性所付出的巨大代价就是其读写性能比较差，**而像微博、facebook这类SNS的应用，对并发读写能力要求极高，关系型数据库已经无法应付(在读方面，传统上为了克服关系型数据库缺陷，提高性能，都是增加一级memcache来静态化网页，而在SNS中，变化太快，memchache已经无能为力了)，因此，必须用新的一种数据结构存储来代替关系数据库。

关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在SNS中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。

于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

## 存储引擎

**数据库**存储引擎是数据库底层软件组织，数据库管理系统（DBMS）使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以 获得特定的功能。

查看支持的存储引擎：show engines;

查看使用的存储引擎：SHOW VARIABLES LIKE 'storage\_engine';

### Myisam

MyISAM表是独立于操作系统的，这说明可以轻松地将其从Windows服务器移植到Linux服务器；Myisam拥有较高的插入 查询速度，但是不支持事务。

这种存储引擎不支持事务，不支持行级锁，只支持并发插入的表锁，主要用于高负载的select。

每当我们建立一个MyISAM引擎的表时，就会在本地磁盘上建立三个文件：1.tb\_demo.frm，存储表结构  
2.tb\_demo.MYD，存储数据  
3.tb\_demo.MYI，存储索引

MyISAM表无法处理事务，这就意味着有事务处理需求的表，不能使用MyISAM存储引擎。

特别适合在以下几种情况下使用：

1.选择密集型的表。MyISAM存储引擎在筛选大量数据时非常迅速，这是它最突出的优点。  
2.插入密集型的表。MyISAM的并发插入特性允许同时选择和插入数据。例如：MyISAM存储引擎很适合管理邮件或Web服务器日志数据。

主要特点：

MyISAM主要特性有：

1、大文件（达到63位文件长度）在支持大文件的文件系统和操作系统上被支持

2、当把删除和更新及插入操作混合使用的时候，动态尺寸的行产生更少碎片。这要通过合并相邻被删除的块，以及若下一个块被删除，就扩展到下一块自动完成

3、每个MyISAM表最大索引数是64，这可以通过重新编译来改变。每个索引最大的列数是16

4、最大的键长度是1000字节，这也可以通过编译来改变，对于键长度超过250字节的情况，一个超过1024字节的键将被用上

5、BLOB和TEXT列可以被索引

6、NULL被允许在索引的列中，这个值占每个键的0~1个字节

7、所有数字键值以高字节优先被存储以允许一个更高的索引压缩

8、**每个MyISAM类型的表都有一个AUTO\_INCREMENT的内部列，当INSERT和UPDATE操作的时候该列被更新，同时AUTO\_INCREMENT列将被刷新。所以说，MyISAM类型表的AUTO\_INCREMENT列更新比InnoDB类型的AUTO\_INCREMENT更快**

9、可以把数据文件和索引文件放在不同目录

10、每个字符列可以有不同的字符集

11、有VARCHAR的表可以固定或动态记录长度

12、VARCHAR和CHAR列可以多达64KB

myisam类型的表支持三种不同的存储结构：静态型、动态型、压缩型。

（1）静态型：就是定义的表列的大小是固定（即不含有：xblob、xtext、varchar等长度可变的数据类型），这样mysql就会自动使用静态myisam格式。

        使用静态格式的表的性能比较高，因为在维护和访问的时候以预定格式存储数据时需要的开销很低。但是这高性能是有空间换来的，因为在定义的时候是固定的，所以不管列中的值有多大，都会以最大值为准，占据了整个空间。

（2）动态型：如果列（即使只有一列）定义为动态的（xblob, xtext, varchar等数据类型），这时myisam就自动使用动态型，虽然动态型的表占用了比静态型表较少的空间，但带来了性能的降低，因为如果某个字段的内容发生改变则其位置很可能需要移动，这样就会导致碎片的产生。随着数据变化的怎多，碎片就会增加，数据访问性能就会相应的降低。

      对于因为碎片的原因而降低数据访问性，有两种解决办法：

     @1、尽可能使用静态数据类型

     @2、经常使用optimize   table语句，他会整理表的碎片，恢复由于表的更新和删除导致的空间丢失。

        (如果存储引擎不支持 optimize  table    则可以转储并重新加载数据，这样也可以减少碎片)

（3）压缩型：如果在这个数据库中创建的是在整个生命周期内只读的表，则这种情况就是用myisam的压缩型表来减少空间的占用。

### Innodb

InnoDB是一个健壮的事务型存储引擎，InnoDB就是作为默认的存储引擎。InnoDB还引入了行级锁定和外键约束。（5.6版本之后对全文索引支持）

Innodb特性：

1、**InnoDB给MySQL提供了具有提交、回滚和崩溃恢复能力的事物安全（ACID兼容）存储引擎。Innodb支持行级锁，有更高的并发性能。**

2、InnoDB是为处理巨大数据量的最大性能设计。它的CPU效率可能是任何其他基于磁盘的关系型数据库引擎锁不能匹敌的

3、InnoDB存储引擎完全与MySQL服务器整合，**I**nnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB将它的表和索引在一个逻辑表空间中，表空间可以包含数个文件（或原始磁盘文件）。这与MyISAM表不同，比如在MyISAM表中每个表被存放在分离的文件中。InnoDB表可以是任何尺寸，即使在文件尺寸被限制为2GB的**操作系统**上

4、**InnoDB支持外键完整性约束，**存储表中的数据时，每张表的存储都按主键顺序存放，如果没有显示在表定义时指定主键，InnoDB会为每一行生成一个6字节的ROWID，并以此作为主键

5、InnoDB被用在众多需要高性能的大型数据库站点上

6、**Innodb支持自增长列（auto\_increment）,自增长列的值不能为空，如果在使用的时候为空的话怎会进行自动存现有的值开始增值，如果有但是比现在的还大，则就保存这个值。**

7、Innodb支持mvcc的行级锁定。

在以下场合下，使用InnoDB是最理想的选择：

1.更新密集的表。InnoDB存储引擎特别适合处理多重并发的更新请求。  
2.事务。InnoDB存储引擎是支持事务的标准MySQL存储引擎。  
3.自动灾难恢复。与其它存储引擎不同，InnoDB表能够自动从灾难中恢复。  
4.外键约束。MySQL支持外键的存储引擎只有InnoDB。  
5.支持自动增加列AUTO\_INCREMENT属性。

一般来说，如果需要事务支持，并且有较高的并发读取频率，InnoDB是不错的选择。

**为什么MyISAM会比Innodb的查询速度快？**

**INNODB在做SELECT的时候，要维护的东西比MYISAM引擎多很多:**  
1）数据块，INNODB要缓存，MYISAM只缓存索引块，  这中间还有换进换出的减少；  
   
2）innodb寻址要映射到块，再到行，MYISAM记录的直接是文件的OFFSET，定位比INNODB要快  
  
3）INNODB还需要维护MVCC一致；虽然你的场景没有，但他还是需要去检查和维护

MVCC (Multi-Version Concurrency Control)多版本并发控制

### **MEMORY**

使用MySQL Memory存储引擎的出发点是速度。为得到最快的响应时间，采用的逻辑存储介质是系统内存。虽然在内存中存储表数据确实会提供很高的性能，但当mysqld守护进程崩溃时，所有的Memory数据都会丢失。获得速度的同时也带来了一些缺陷。它要求存储在Memory数据表里的数据使用的是长度不变的格式，这意味着不能使用BLOB和TEXT这样的长度可变的数据类型，VARCHAR是一种长度可变的类型，但因为它在MySQL内部当做长度固定不变的CHAR类型，所以可以使用。

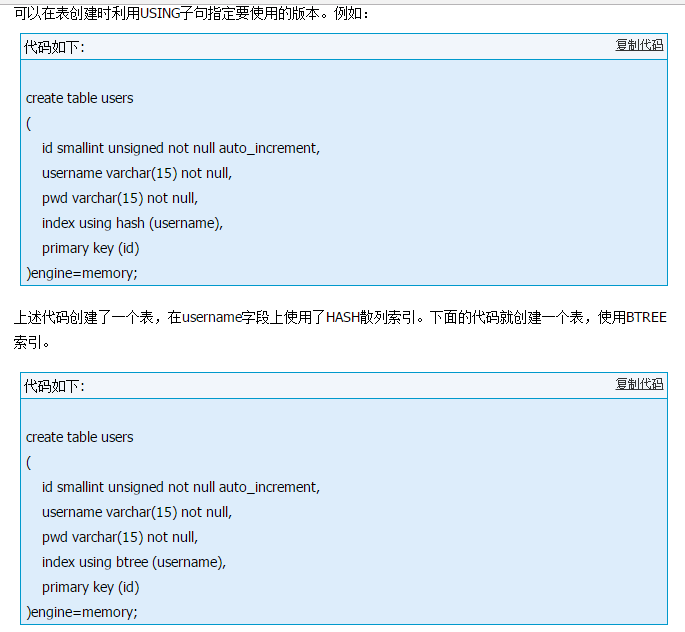
一般在以下几种情况下使用Memory存储引擎：

1.目标数据较小，而且被非常频繁地访问。在内存中存放数据，所以会造成内存的使用，可以通过参数max\_heap\_table\_size控制Memory表的大小，设置此参数，就可以限制Memory表的最大大小。

2.如果数据是临时的，而且要求必须立即可用，那么就可以存放在内存表中。

3.存储在Memory表中的数据如果突然丢失，不会对应用服务产生实质的负面影响。

Memory同时支持散列索引和B树索引。B树索引的优于散列索引的是，可以使用部分查询和通配查询，也可以使用<、>和>=等操作符方便数据挖掘。散列索引进行“相等比较”非常快，但是对“范围比较”的速度就慢多了，因此散列索引值适合使用在=和<>的操作符中，不适合在<或>操作符中，也同样不适合用在order by子句中。



### **MERGE**

MERGE存储引擎是一组MyISAM表的组合，这些MyISAM表结构必须完全相同，尽管其使用不如其它引擎突出，但是在某些情况下非常有用。说白了，Merge表就是几个相同MyISAM表的聚合器；Merge表中并没有数据，对Merge类型的表可以进行查询、更新、删除操作，这些操作实际上是对内部的MyISAM表进行操作。Merge存储引擎的使用场景。

对于服务器日志这种信息，一般常用的存储策略是将数据分成很多表，每个名称与特定的时间端相关。例如：可以用12个相同的表来存储服务器日志数据，每个表用对应各个月份的名字来命名。当有必要基于所有12个日志表的数据来生成报表，这意味着需要编写并更新多表查询，以反映这些表中的信息。与其编写这些可能出现错误的查询，不如将这些表合并起来使用一条查询，之后再删除Merge表，而不影响原来的数据，删除Merge表只是删除Merge表的定义，对内部的表没有任何影响。

## Limit分页查询优化

Mysql分页查询比较简单，直接limit offset,length可以获取数据，但是当offset和length比较大的时候，mysql性能会下降。

比如select \* from tableA limit 100000,10;mysql在查询的时候会把100000之前的所有数据都会扫描一次，然后丢弃，最后取出100000后的10条数据。

### 子查询优化

**利用主键id的索引到起始位置，然后再截取需要的行数。**

Select \* from tableA where id>100000 limit 10;

先找出第一条数据，然后大于等于这条数据的 id就是要获取的数据

缺点：数据必须是连续的，不能有where条件，因为where条件会筛选数据，导致数据失去连续性。

Select \* from tableA where id>=(select id from tableA limit 100000,1) limit 10;

### 倒排表优化

类似建立索引，用一张表维护页数，然后通过高效的连接得到数据。

缺点：只适合数据数固定的情况，数据不能删除，维护页表困难。

倒排表是指存放在内存中的能够追加倒排记录的倒排索引。倒排表是迷你的倒排索引。  
临时倒排文件是指存放在磁盘中，以文件的形式存储的不能够追加倒排记录的倒排索引。临时倒排文件是中等规模的倒排索引。  
最终倒排文件是指由存放在磁盘中，以文件的形式存储的临时倒排文件归并得到的倒排索引。最终倒排文件是较大规模的倒排索引。  
倒排索引作为抽象概念，而倒排表、临时倒排文件、最终倒排文件是倒排索引的三种不同的表现形式。

### 反向查找优化

当偏移超过一半记录数的时候，先用排序，这样偏移就反转了。

缺点：order by优化比较麻烦，要增加索引，索引影响数据的修改效率，并且要知道总记录数。

Limit偏移算法：

正向查找：（当前页-1）\*页长度

反向查找：总记录-当前页\*页长度

SELECT \* FROM `abc` WHERE `BatchID` = 123 LIMIT 1199960, 40

SELECT \* FROM `abc` WHERE `BatchID` = 123 ORDER BY InputDate DESC LIMIT 428775, 40

### 只查索引法

MySQL的limit工作原理就是先读取n条记录，然后抛弃前n条，读m条想要的，所以n越大，性能会越差。

SELECT \* FROM member ORDER BY last\_active LIMIT 50,5

SELECT \* FROM member INNER JOIN (SELECT member\_id FROM member ORDER BY last\_active LIMIT 50, 5) USING (member\_id)

区别在于：优化前的sql需要更多的IO资源，因为先读索引再读数据，然后抛弃不需要的行。而优化后的只读索引就可以了，然后通过索引读取需要的列。

总结：当一个数据库表过于庞大，LIMIT offset, length中的offset值过大，则SQL查询语句会非常缓慢，你需增加order by，并且order by字段需要建立索引。   
如果使用子查询去优化LIMIT的话，则子查询必须是连续的，某种意义来讲，子查询不应该有where条件，where会过滤数据，使数据失去连续性。   
如果你查询的记录比较大，并且数据传输量比较大，比如包含了text类型的field，则可以通过建立子查询。  
SELECT id,title,content FROM items WHERE id IN (SELECT id FROM items ORDER BY id limit 900000, 10);   
如果limit语句的offset较大，你可以通过传递pk键值来减小offset = 0，这个主键最好是int类型并且auto\_increment  
SELECT \* FROM users WHERE uid > 456891 ORDER BY uid LIMIT 0, 10;  
这条语句，大意如下:  
SELECT \* FROM users WHERE uid >=  (SELECT uid FROM users ORDER BY uid limit 895682, 1) limit 0, 10;   
如果limit的offset值过大，用户也会翻页疲劳，你可以设置一个offset最大的，超过了可以另行处理，一般连续翻页过大，用户体验很差，则应该提供更优的用户体验给用户。

## 数据库和实例

数据库：物理操作系统文件或者磁盘的集合。

实例：一组后台进程/线程以及一个内存共享区，这些内存是由同一个计算机上运行的线程/进程所共享。

数据库可以由多个实例装载和打开。而实例可以在任何时间点装载和打开一个数据库。实际上，准确地讲，实例在其整个生存期中最多能装载和打开一个数据库！

MySQL是单进程多线程（而Oracle等是多进程），也就是说MySQL实例在系统上表现就是一个服务进程，即进程（通过多种方法可以创建多实例，再安装一个端口号不同的[mysql](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=0&is_app=0&jk=11f201f6e90a0102&k=mysql&k0=mysql&kdi0=0&luki=8&mcpm=0&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=2010ae9f601f211&ssp2=1&stid=9&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fdb%2Fmysql%2F201503%2F97575%2Eshtml&urlid=0)，或者通过workbench来新建一个端口号不同的服务器实例等）。

MySQL实例是线程和内存组成，实例才是真正用于操作数据库文件的（MySQL数据库是由一些列物理文件组成，类似于frm、MYD、MYI、ibd结尾的文件）；

一般情况下一个实例操作一个或多个数据库（Oracle一个实例对应一个数据库）；集群情况下多个实例操作一个或多个数据库。

不同实例之间使用不同的端口号进行区分，各个实例数据可以使用不同的磁盘目录。

MySQL多实例通过mysqld\_multi工具来进行管理。多个实例使用的是同一个配置文件my.cnf，在配置文件里面配置不同的端口号即可。然后通过mysqld\_multi start+端口号启动。

mysql中database、instance、session：

**MySQL**中建立一个会话，不是和具体的**数据库**相连接，而是跟某个instance建立会话（每个会话可以使用不同的用户身份）。

      而一个实例可以操作多个数据库，故一个会话（在**操作系统**概念里，会话即是线程）可以操作一个实例上的多个数据库。

MySQL运行多实例有2种方法，第一种是使用多个配置文件启动不同的进程来实现多实例;第二种是通过mysqld\_multi使用单独的配置文件来实现多实例

## 三大范式

### 第一范式1NF

数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

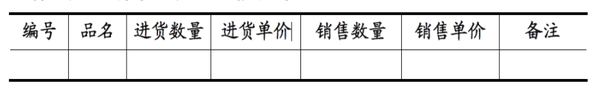
如果出现重复的属性，就需要定义一个新的实体，新的实体由重复的属性构成，新的实体和原实体之间是一对多关系。

在任何一个关系型数据库中，第一范式是基本的要求，必须满足。

下面看例子：



上面的数据表是不符合第一范式的，因为进货下面又可以再次划分为数量和单价，销售同理。满足第一范式的数据表应设计如下：



仅仅满足第一范式的数据库表还会存在很多问题，如数据冗余过大，插入异常，删除异常，修改异常。看下表；



存在的问题：

1 每一名学生的学号 姓名 系名 系主任这些数据重复过多，每个系与系主任的数据也重复多次——数据冗余

2 假设学校新开了一个系，但是这个系还没有开始招生，那么现在是无法将系主任和系名的数据添加到上述的表中——插入异常

3 假设将某个系中的所有学生的相关记录全部删除，那么系和系主任的数据也随之消失了，但是学生记录不存在，并不意味着这个系就没有了——删除异常

4 假设小明转到哲学系，那么为了保证数据库的一致性，需要修改三条记录中系与系主人的数据——修改异常

因此需要提高设计标准，使其符合更高一级的范式2NF，即“规范化”。

### 第二范式2NF

第二范式是在第一范式基础上建立起来的，即必须先满足第一范式。第二范式要求数据库表中的每个实例或者行必须可以被唯一的区分。为实现区分，通常需要为每个表加上一个列，以存储各个实例的唯一标识。这个唯一的属性列称为主键字或主键 主码。

第二范式要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体和原实体之间是一对多的关系。简而言之，就是非主属性非部分依赖于主关键字。

如下表：

学生 课程 教师 教师职称 教材 教室 上课时间  
李四 Spring 张老师 java讲师 《Spring深入浅出》 301 08:00  
张三 Struts 杨老师 java讲师 《Struts in Action》 302 13:30

这里通过（学生 课程）可以确定一条记录。故将其作为联合主键，但是教材并不完全依赖于（学生 课程），只拿出部分课程就可以确定教材。这是部分依赖。

正确的表应设计如下：

选课表

学生 课程 教师 教师职称 教室 上课时间  
李四 Spring 张老师 java讲师 301 08:00  
张三 Struts 杨老师 java讲师 302 13:30

课程包：

课程 教材   
Spring 《Spring深入浅出》   
Struts 《Struts in Action》

第二范式消除了部分依赖，减少了插入异常 删除异常和修改异常。

如果继续以第一范式中那张表为例，（学号 课名）作为主键才能唯一标识分数，但是其他的属性如姓名 系名和系主任只需要学号就可以唯一确定了，这也是部分依赖了。

将表修改如下：



是否还存在之前的问题？

1 小明转到法律系，只需要修改一次对应的系的值即可。

2 姓名 系名 系主任没有重复那么多次，数据冗余减少了。

3 删除某个系中学生记录，信息仍然全部丢失。

4 插入一个尚无学生的新系信息，仍然存在插入异常。

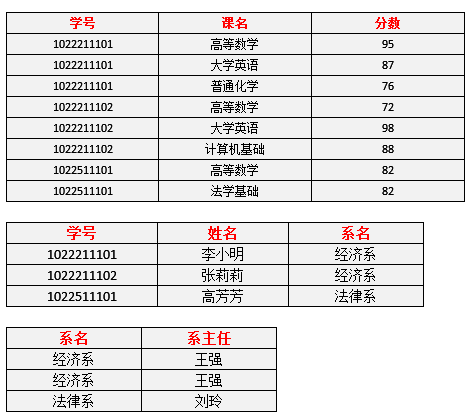
### 第三范式3NF

满足第三范式必须先满足第二范式。第三范式要求一个数据表中不包含已在其他表中包含的非主关键字信息。例如一个部门信息表，每个部门有个部门编号，部门名称等，那么在员工信息表中列出部门编号后，就不能再将部门名称等与部门相关的信息再加入到员工信息表中。如果不存在部门信息表，那根据第三范式也应该构建他，否则就会出现大量的冗余。简而言之，第三范式就是属性不依赖于其他非主属性。（消除传递依赖，消除冗余）。

仍以上述表为例，对于学生表，系名决定了系主任，存在了非主属性系主任对学号的传递函数依赖，不符合3NF。

故对表进行进一步分解：

选课（学号，课名，分数）  
学生（学号，姓名，系名）  
系（系名，系主任）



上述的问题也都得到了解决。

由此可见，3NF基本上解决了数据冗余过大，插入异常，修改异常，删除异常的问题。

总结：1NF就是原子性，字段不可再分割

2NF就是完全依赖，非主属性完全依赖于主属性，不存在部分依赖

3NF就是非主属性中没有传递依赖。每列中都与主键有直接关系。

## 数据库优化思路

什么是优化？

合理安排资源 调整系统参数 使mysql运行更快 更节省资源

优化是多方面的，包括查询 更新 服务器等。

原则：减少系统瓶颈 减少资源占用 增加系统反应速度

### 1.SQL语句优化

1）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。  
2）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  
**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  
3）很多时候用 exists 代替 in 是一个好的选择  
4）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤

5）一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

6）select \* from users where YEAR(adddate)<2007;  
将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成  
select \* from users where adddate<‘2007-01-01';

7）使用连接join代替子查询

使用SELECT语句来创建一个单列的查询结果，然后把这个结果作为过滤条件用在另一个查询中。

例如，我们要将客户基本信息表中没有任何订单的客户删除掉，就可以利用子查询先从销售信息表中将所有发出订单的客户ID取出来，然后将结果传递给主查询：

DELETE FROM customerinfo

WHERE CustomerID NOT in (SELECT Customer ID FROM salesinfo)

使用子查询可以一次性的完成很多逻辑上需要多个步骤才能完成的SQL操作，同时也可以避免事务或者表锁死，并且写起来也很容易。但是，有些情况下，子查询可以被更有效率的连接（JOIN）..替代。

例如，假设我们要将所有没有订单记录的用户取出来，可以用下面这个查询完成：

SELECT \* FROM customerinfo

WHERE CustomerID NOT in (SELECT CustomerID FROM salesinfo)

如果使用连接（JOIN）..来完成这个查询工作，速度将会快很多。尤其是当salesinfo表中对CustomerID建有索引的话，性能将会更好，查询如下：

SELECT \* FROM customerinfo

LEFT JOIN salesinfoONcustomerinfo.CustomerID=salesinfo.CustomerID

WHERE salesinfo.CustomerID IS NULL

连接（JOIN）..之所以更有效率一些，是因为MySQL不需要在内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。

8）查询语句的查询条件中只有OR关键字，且OR前后的两个条件中的列都是索引时，索引才会生效，否则，索引不生效。

### 2.索引优化

#### 创建索引

对于查询为主要的应用，如果不添加索引，在进行查询的时候都会进行一次全表扫描。如果数据量很大而符合条件的结果又比较少，那就必须加索引。但是对于比如性别就两个取值，建立索引没有什么优势，还会影响更新的速度。

#### 复合索引

比如有一条语句是这样的：select \* from users where area='beijing' and age=22;  
如果我们是在area和age上分别创建单个索引的话，由于mysql查询每次只能使用一个索引，所以虽然这样已经相对不做索引时全表扫描提高了很多效率，但是如果在area、age两列上创建复合索引的话将带来更高的效率。如果我们创建了(area, age, salary)的复合索引，那么其实相当于创建了(area,age,salary)、(area,age)、(area)三个索引，这被称为最佳左前缀特性。因此我们在创建复合索引时应该将最常用作限制条件的列放在最左边，依次递减。

MySQL可以为多个字段创建索引，一个索引可以包括16个字段。**对于联合索引，只有查询条件中使用了这些字段中第一个字段时，索引才会生效。**

#### 索引不会包含有NULL值的列

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

#### 使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的 列，如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

#### 排序的索引问题

mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

### 子查询优化

MySQL从4.1版本开始支持子查询，使用子查询进行SELECT语句嵌套查询，可以一次完成很多逻辑上需要多个步骤才能完成的SQL操作。

子查询虽然很灵活，但是执行效率并不高。

执行子查询时，MYSQL需要创建临时表，查询完毕后再删除这些临时表，所以，子查询的速度会受到一定的影响。

优化：

可以使用连接查询（JOIN）代替子查询，连接查询时不需要建立临时表，其速度比子查询快。

### 3.数据库结构优化

1）范式优化： 比如消除冗余（节省空间。。） 2）反范式优化：比如适当加冗余等（减少join） 3）拆分表： 分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘I/O，一个精心设置的分区可以将数据传输对磁盘I/O竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。  
4）拆分其实又分垂直拆分和水平拆分： 案例： 简单购物系统暂设涉及如下表： 1.产品表（数据量10w，稳定） 2.订单表（数据量200w，且有增长趋势） 3.用户表 （数据量100w，且有增长趋势） 以[mysql](http://www.2cto.com/database/MySQL/)为例讲述下水平拆分和垂直拆分，mysql能容忍的数量级在百万静态数据可以到千万 **垂直拆分：**解决问题：表与表之间的io竞争 不解决问题：单表中数据量增长出现的压力 方案： 把产品表和用户表放到一个server上 订单表单独放到一个server上 **水平拆分：** 解决问题：单表中数据量增长出现的压力 不解决问题：表与表之间的io争夺  
方案： 用户表通过性别拆分为男用户表和女用户表 订单表通过已完成和完成中拆分为已完成订单和未完成订单 产品表 未完成订单放一个server上 已完成订单表盒男用户表放一个server上 女用户表放一个server上(女的爱购物 哈哈)

#### 将字段很多的表分解成多个表

对于字段较多的表，如果有些字段的使用频率很低，可以将这些字段分离出来形成新表。

因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。

（比如用户信息表，可以分为基本信息表和更多信息表，将不常用的信息放在另外一个表中）

#### 增加中间表

对于需要经常联合查询的表，可以建立中间表以提高查询效率。

通过建立中间表，将需要通过联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询。

（比如对5张表进行left join，每次查询都需要连接这5张表，建立一个中间表，将查询的结果放在中间表中，下次直接从中间表中查询出来。）

应用：常用于统计中。在统计报表 统计结果时，一般都是离线统计。后台进程或者线程进行统计计算，放在中间表中，在页面中显示时，直接对这个表查询）

#### 增加冗余字段

设计数据表时应尽量遵循范式理论的规约，尽可能的减少冗余字段，让数据库设计看起来精致、优雅。但是，合理的加入冗余字段可以提高查询速度。

表的规范化程度越高，表和表之间的关系越多，需要连接查询的情况也就越多，性能也就越差。

**注意：**

**冗余字段的值在一个表中修改了，就要想办法在其他表中更新，否则就会导致数据不一致的问题。**

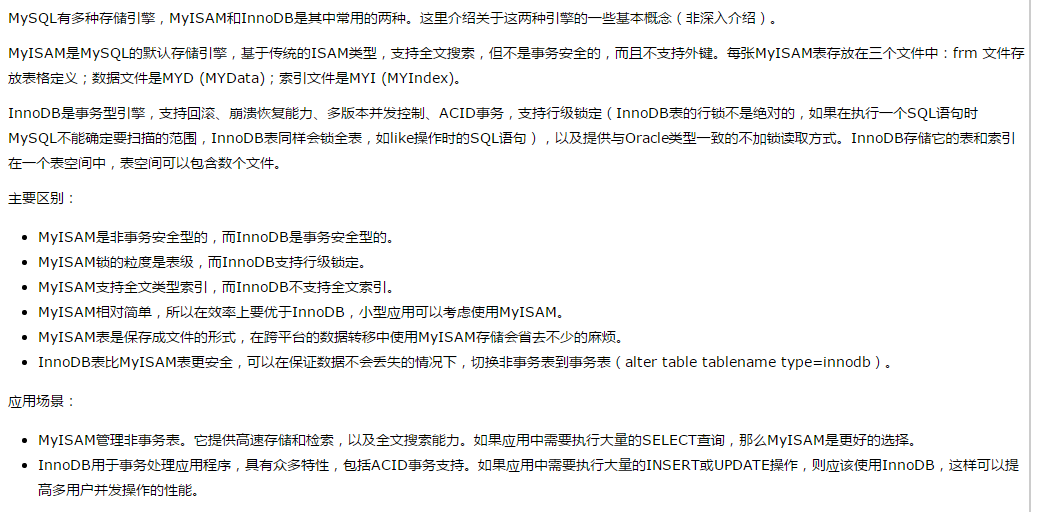
应用：比如在购物车中，存储了商品的id和商品的标题，商品主图，商品价格。这里可以只存储商品id，然后去商品表中查询。但是为了查询速度，这里使用了冗余。

### 插入数据的优化

插入数据时，影响插入速度的主要是索引、唯一性校验、一次插入的数据条数等。

（索引越多，查询越快，但是写入数据越慢，因此在写数据时，不仅要把数据写到文件中，而且还要更新索引文件中的数据）

插入数据的优化，不同的存储引擎优化手段不一样，在MySQL中常用的存储引擎有，MyISAM和InnoDB，两者的区别：



#### Myisam

##### 禁用索引

对于非空表，插入记录时，MySQL会根据表的索引对插入的记录建立索引。如果插入大量数据，建立索引会降低插入数据速度。

为了解决这个问题，可以在批量插入数据之前禁用索引，数据插入完成后再开启索引。

禁用索引的语句：

*ALTER TABLE table\_name DISABLE KEYS*

开启索引语句：

*ALTER TABLE table\_name ENABLE KEYS*

对于空表批量插入数据，则不需要进行操作，因为MyISAM引擎的表是在导入数据后才建立索引。

##### 禁用唯一性检查

唯一性校验会降低插入记录的速度，可以在插入记录之前禁用唯一性检查，插入数据完成后再开启。

禁用唯一性检查的语句：*SET UNIQUE\_CHECKS = 0;*

开启唯一性检查的语句：SET UNIQUE\_CHECKS = 1;

##### 批量插入数据

插入数据时，可以使用一条INSERT语句插入一条数据，也可以插入多条数据。





第二种方式的插入速度比第一种方式快。

Mysql会解析sql语句，第一种方式需要解析两次，第二种方式只需要解析一次。

##### 使用LOAD DATA INFILE

当需要批量导入数据时，使用LOAD DATA INFILE语句比INSERT语句插入速度快很多

#### Innodb

##### 禁用唯一性检查

和myisam表一样

##### 禁用外键检查

插入数据之前执行禁止对外键的检查，数据插入完成后再恢复，可以提供插入速度。

禁用：SET foreign\_key\_checks = 0;

开启：SET foreign\_key\_checks = 1;

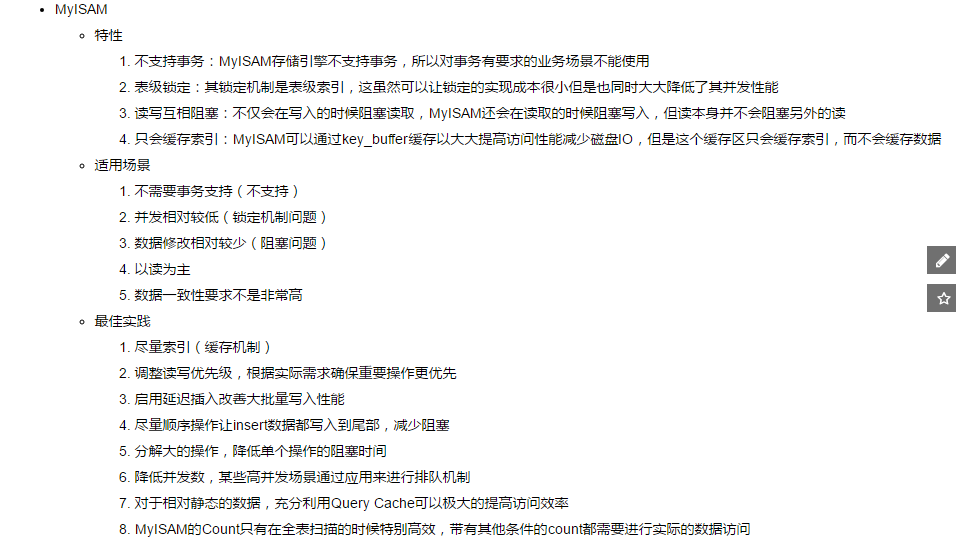
##### 禁止自动提交

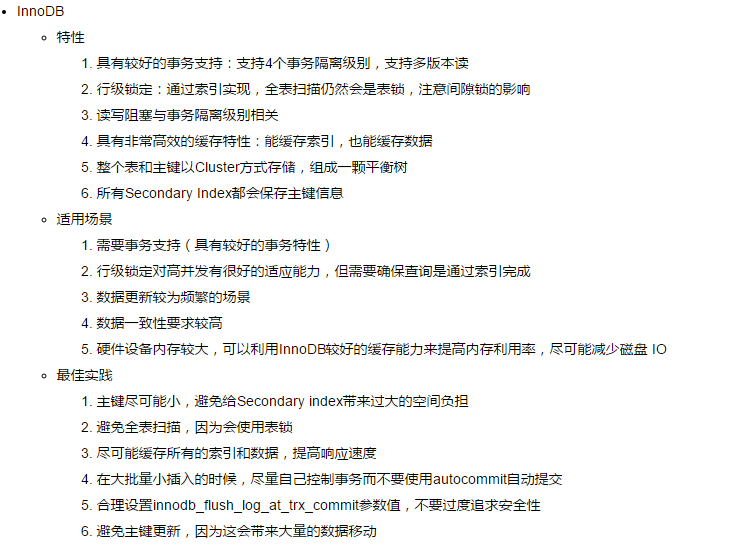
插入数据之前执行禁止事务的自动提交，数据插入完成后再恢复，可以提高插入速度。

禁用：SET autocommit = 0;

开启：SET autocommit = 1;

### 4.存储引擎优化





### 5.服务器优化

#### 硬件优化

服务器的硬件性能直接决定着MySQL数据库的性能，硬件的性能瓶颈，直接决定MySQL数据库的运行速度和效率。

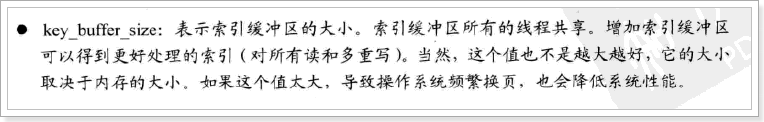
需要从以下几个方面考虑：

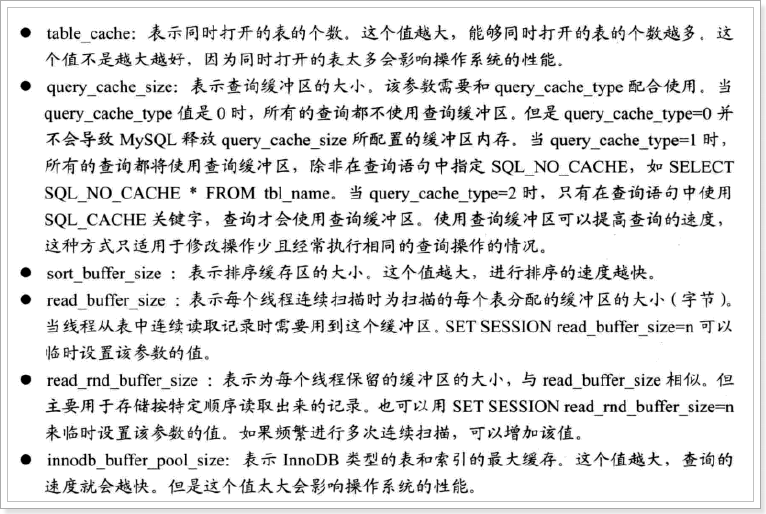
1. 配置较大的内存。足够大的内存，是提高MySQL数据库性能的方法之一。内存的IO比硬盘快的多，可以增加系统的缓冲区容量，使数据在内存停留的时间更长，以减少磁盘的IO。
2. 配置高速磁盘，比如SSD。
3. 合理分配磁盘IO，把磁盘IO分散到多个设备上，以减少资源的竞争，提高并行操作能力。
4. 配置多核处理器，MySQL是多线程的数据库，多处理器可以提高同时执行多个线程的能力。

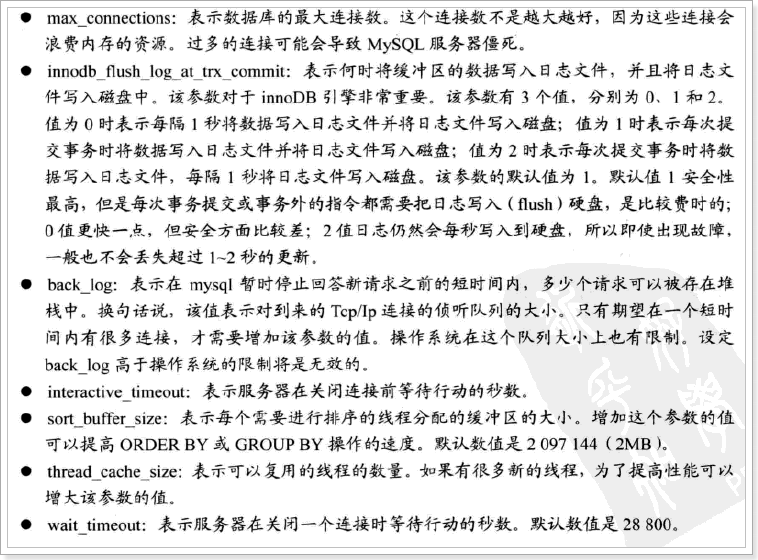
#### 优化mysql参数

通过优化MySQL的参数可以提高资源利用率，从而达到提高MySQL服务器性能的目的。

MySQL的配置参数都在my.conf或者my.ini文件的[mysqld]组中，常用的参数如下：







要求：必须记忆至少3个。

## 索引

索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。更通俗的说，数据库索引好比是一本书前面的目录，能加快数据库的查询速度。上述SQL语句，在没有索引的情况下，数据库会遍历全部200条数据后选择符合条件的；而有了相应的索引之后，数据库会直接在索引中查找符合条件的选项。如果我们把SQL语句换成“SELECT \* FROM article WHERE id=2000000”，那么你是希望数据库按照顺序读取完200万行数据以后给你结果还是直接在索引中定位呢？

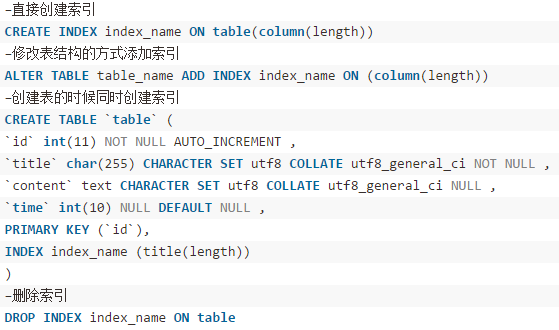
一般数据库默认都会为主键生成索引。

索引分为聚簇索引和非聚簇索引两种，聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

### ****索引类型****

#### 普通索引

最基本的索引。MyIASM中默认的BTREE类型的索引，也是我们大多数情况下用到的索引。

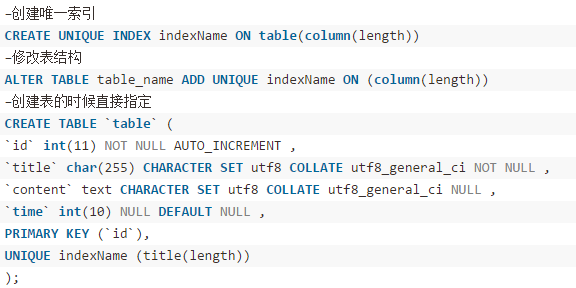


CREATE UNIQUE INDEX index\_name

ON table\_name (column\_name)

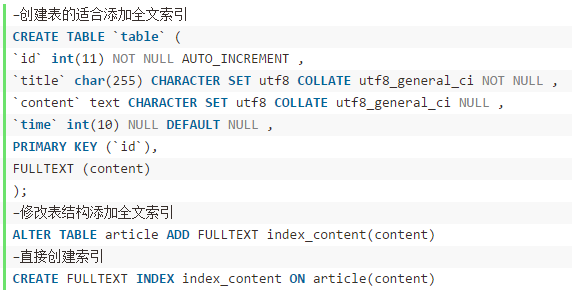
#### 唯一索引

与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。



#### 全文索引

Mysql5.6后的Innodb存储引擎也支持了全文索引。对于大容量的数据表，生成全文索引是一个非常消耗时间非常消耗硬盘空间的做法。



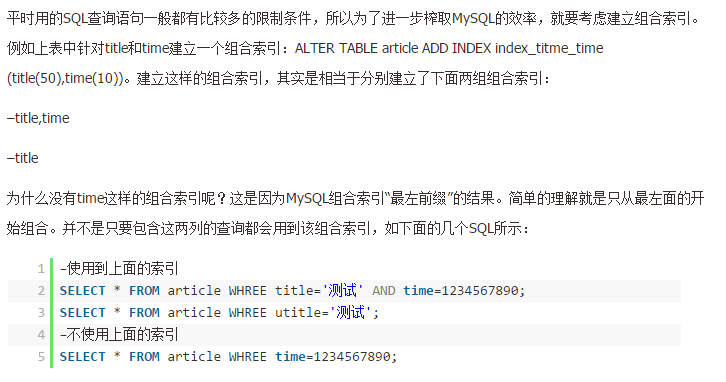
ULLTEXT 用于搜索很长一篇文章的时候，效果最好。用在比较短的文本，如果就一两行字的，普通的 INDEX 也可以。

#### 单列 多列索引

多个单列索引与单个多列索引的查询效果不同，因为执行查询时，MySQL只能使用一个索引，会从多个索引中选择一个限制最为严格的索引。

组合索引属于多列索引。

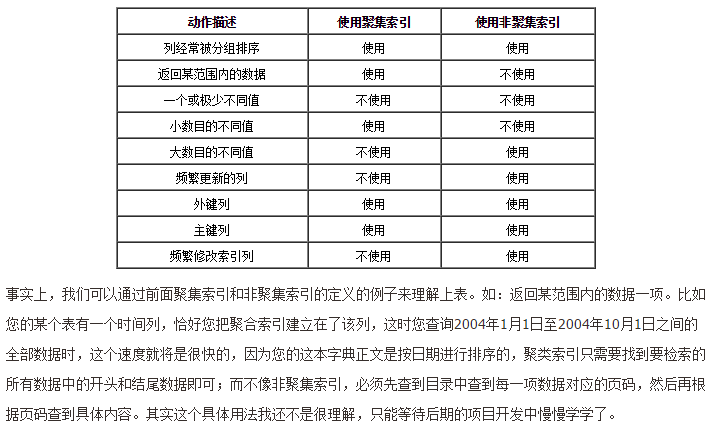
#### 组合索引（最左前缀）



### ****索引优化****

**索引可以大大提高查询速度，但是同时也会降低更新表的速度。如对表进行insert update delete操作。**因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。建立索引会占用磁盘空间的索引文件。如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。

#### 聚集索引和非聚集索引



#### 索引中不能含有null值的列

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

#### 使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

#### 索引列排序

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

#### Like语句操作

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

#### 不要在列上进行运算

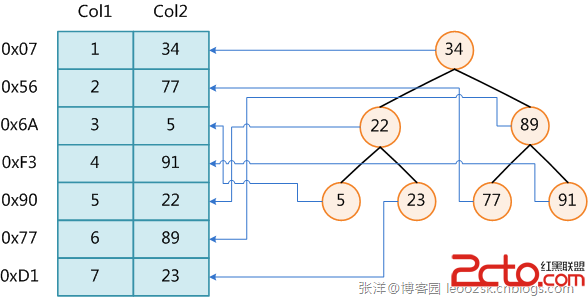
select \* from users where YEAR(adddate)<2007，将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成：select \* from users where adddate<’2007-01-01′。

MySQL只对一下操作符才使用索引：<,<=,=,>,>=,between,in,以及某些时候的like(不以通配符%或\_开头的情形)。而理论上每张表里面最多可创建16个索引。

**数据库索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

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。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



图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)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在数据库设计器中创建三种索引：**唯一索引、主键索引和聚集索引**。

**唯一索引**

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。 **主键索引** 数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。 **聚集索引** 在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。**一个表只能包含一个聚集索引。**

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。**与非聚集索引相比，聚集索引通常提供更快的数据访问速度。**

### ****索引底层原理****

<http://www.cnblogs.com/shijingxiang/articles/4743324.html>

<http://blog.csdn.net/cangchen/article/details/44818485>

平衡二叉搜索树：每个节点只有两个子节点，当数据量非常多时，树的高度就会增加，查找性能会差很多，不适合做大量数据存储的基础结构。

B树：也是B-树，是一棵平衡的m路搜索树。每个非根节点包含的关键字个数j满足

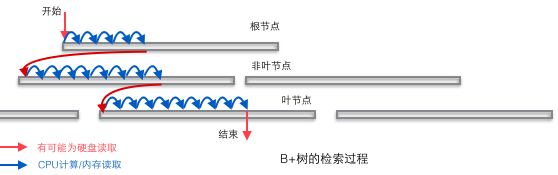
┌m/2┐ - 1 <= j <= m – 1 一个子节点的数量比关键字个数多1.这样关键字就变成了子节点的分割标志。

数据同时存放在叶子节点和非叶子节点，无法简单完成按顺序遍历B树中的关键字，必须用中序遍历方法。

B+树：一棵m阶B+树是一棵平衡的m路搜索树。最重要的性质是每个非根节点所包含的关键字个数 j 满足：┌m/2┐ - 1 <= j <= m；子树的个数最多可以与关键字一样多。非叶节点存储的是子树里最小的关键字。同时数据节点只存在于叶子节点中，且叶子节点间增加了横向的指针，这样顺序遍历所有数据将变得非常容易。

 B\*树：一棵m阶B树是一棵平衡的m路搜索树。最重要的两个性质是1每个非根节点所包含的关键字个数 j 满足：┌m2/3┐ - 1 <= j <= m；2非叶节点间添加了横向指针。

B+树适合作为数据库的基础结构，完全是因为计算机的内存-机械硬盘两层存储结构。内存可以完成快速的随机访问（随机访问即给出任意一个地址，要求返回这个地址存储的数据）但是容量较小。而硬盘的随机访问要经过机械动作（1磁头移动 2盘片转动），访问效率比内存低几个数量级，但是硬盘容量较大。典型的数据库容量大大超过可用内存大小，这就决定了在B+树中检索一条数据很可能要借助几次磁盘IO操作来完成。如下图所示：通常向下读取一个节点的动作可能会是一次磁盘IO操作，不过非叶节点通常会在初始阶段载入内存以加快访问速度。同时为提高在节点间横向遍历速度，真实数据库中可能会将图中蓝色的CPU计算/内存读取优化成二叉搜索树（InnoDB中的page directory机制）。



真实数据库中的B+树应该是非常扁平的

对于只有单关键字组成的表，只需要将关键字存储到树的节点即可。当数据库一条记录里包含多个字段时，一棵B+树就只能存储主键，如果检索的是非主键字段，那么主键索引就会失去作用，会进行全表扫描，顺序查找，此时应该在检索的列上建立第二套索引。

有两种方法可以解决多个B+树访问同一套表数据的问题，一种是聚簇索引，一种是非聚簇索引。

对于聚簇索引，数据和主键B+树存储在一起，辅助键B+树存储的是辅助键和主键，主键B+树和辅助键B+树几乎是两种类型的树。

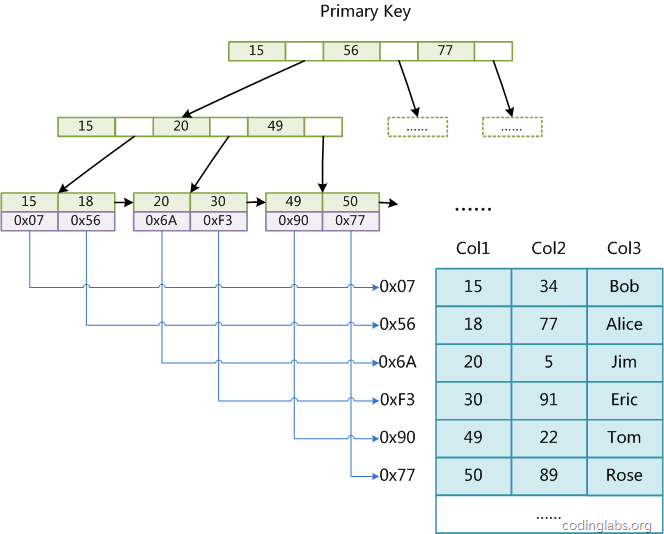
对于非聚簇索引，主键B+树在叶子节点指向真正数据行的指针，而非主键。

InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用"where id = 14"这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。若对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。第二步使用主键在主索引B+树种再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。

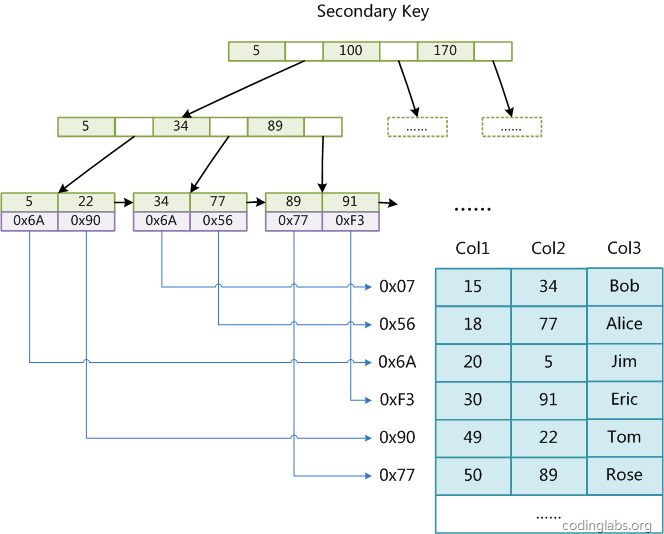
MyISM使用的是非聚簇索引，非聚簇索引的两棵B+树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

Myisam：

主键索引：



辅助索引：

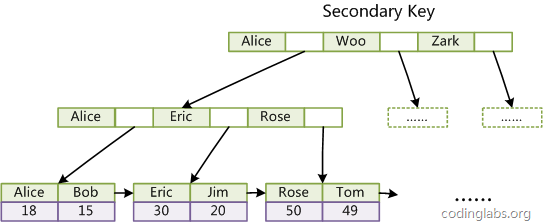


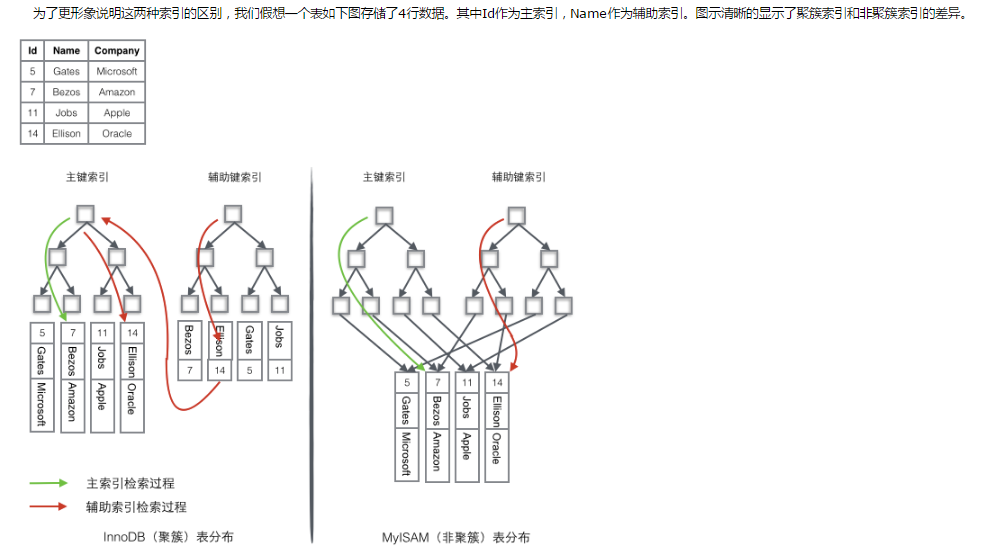
Innodb：

主键索引：



辅助索引：





我们重点关注聚簇索引，看上去聚簇索引的效率明显要低于非聚簇索引，因为每次使用辅助索引检索都要经过两次B+树查找，这不是多此一举吗？聚簇索引的优势在哪？

　　1 由于行数据和叶子节点存储在一起，这样主键和行数据是一起被载入内存的，找到叶子节点就可以立刻将行数据返回了，如果按照主键Id来组织数据，获得数据更快。

2 辅助索引使用主键作为"指针" 而不是使用地址值作为指针的好处是，减少了当出现行移动或者数据页分裂时辅助索引的维护工作，使用主键值当作指针会让辅助索引占用更多的空间，换来的好处是InnoDB在移动行时无须更新辅助索引中的这个"指针"。也就是说行的位置（实现中通过16K的Page来定位，后面会涉及）会随着数据库里数据的修改而发生变化（前面的B+树节点分裂以及Page的分裂），使用聚簇索引就可以保证不管这个主键B+树的节点如何变化，辅助索引树都不受影响。

B+树索引并不能找到一个给定键值的具体行。B+树索引只能找到的是**被查找数据行所在的页**。然后数据库通过把**页**读入内存，再在内存中进行查找，最后得到查找的数据。

Page：是InnoDB磁盘管理的最小单位，与数据库相关的所有内容都存储在这种Page结构里。

Page分为数据页

**Undo页**：为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为Undo Log）。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。除了可以保证事务的原子性，Undo Log也可以用来辅助完成事务的持久化。）还用Undo Log来实现多版本并发控制(简称：MVCC)

**Redo页**：Redo Log记录的是**新数据**的备份。在事务提交前，只要将Redo Log持久化即可，  
不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据Redo Log的内容，将所有数据恢复到最新的状态。

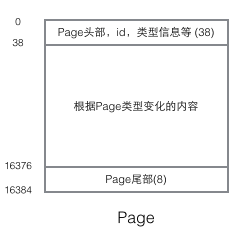
**Undo + Redo事务的简化过程**  
  假设有A、B两个数据，值分别为1,2.  
  A.事务开始.  
  B.记录A=1到undo log.  
  C.修改A=3.  
  D.记录A=3到redo log.  
  E.记录B=2到undo log.  
  F.修改B=4.  
  G.记录B=4到redo log.  
  H.将redo log写入磁盘。  
  I.事务提交

**A-G的过程是在内存中进行的，相应的操作记录在redo log buffer（B&E），redo log buffer（E&G）中，事务执行结果（此时未提交）也存在db buffer中（C&F），buffer满了就写入磁盘当中，如果buffer存储的事务数量都是1个，也就意味着是将日志立即刷入磁盘，那么数据的一致性很好保证。如果存储多个的话，是一次事务完成就会先将redo log同步到磁盘当中并有一个状态位来记录是否提交，再去真正的提交事务，将db buffer 中的数据同步到DB的磁盘当中去。要保证在db buffer中的内容写入磁盘**[**数据库**](http://lib.csdn.net/base/mysql)**文件之前，应当把log buffer的内容写入磁盘日志文件。这种方式可以减少磁盘IO，增加吞吐量。 不过，这种方式适用于一致性要求不高的情景。因为如果出现断电等系统故障，log buffer、db buffer中的完成的事务还没同步到磁盘会丢失。 像银行这种要求事务较高的一致性，就一定要保证每次事务都要记录到磁盘中，如果服务器down了的时候去redo log中恢复，重做一次已经提交的事务。**

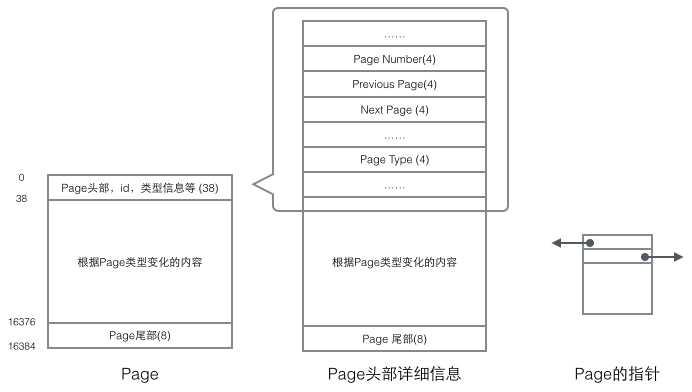
**Undo + Redo事务的特点**  
  A. 为了保证持久性，必须在事务提交前将Redo Log持久化。  
  B. 数据不需要在事务提交前写入磁盘，而是缓存在内存中。  
  C. Redo Log 保证事务的持久性。  
  D. Undo Log 保证事务的原子性。  
  E. 有一个隐含的特点，数据必须要晚于redo log写入持久存储。

**系统页（System Page） 事务数据页（Transaction System Page）**

单个Page的大小是16K（编译宏UNIV\_PAGE\_SIZE控制），每个Page使用一个32位的int值来唯一标识，这也正好对应InnoDB最大64TB的存储容量（16Kib \* 2^32 = 64Tib）。

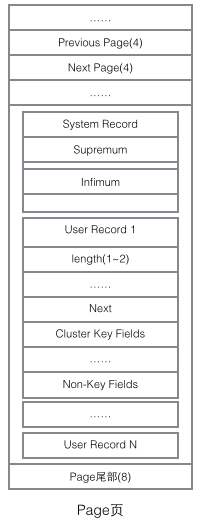


Page头部中的内容

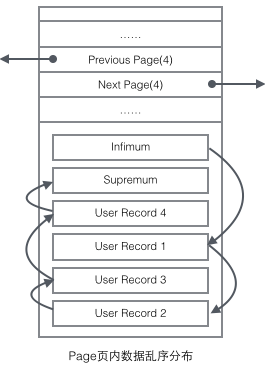


我们重点关注和数据组织结构相关的字段：Page的头部保存了两个指针，分别指向前一个Page和后一个Page，头部还有Page的类型信息和用来唯一标识Page的编号。根据这两个指针我们很容易想象出Page链接起来就是一个双向链表的结构。

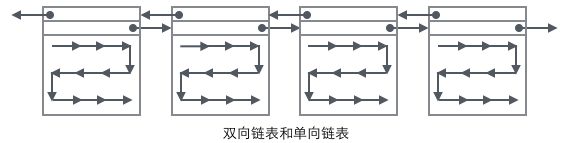
再看看Page的主体内容，我们主要关注行数据和索引的存储，他们都位于Page的User Records部分，User Records占据Page的大部分空间，User Records由一条一条的Record组成，每条记录代表索引树上的一个节点（非叶子节点和叶子节点）。在一个Page内部，单链表的头尾由固定内容的两条记录来表示，字符串形式的"Infimum"代表开头，"Supremum"代表结尾。这两个用来代表开头结尾的Record存储在System Records的段里，这个System Records和User Records是两个平行的段。InnoDB存在4种不同的Record，它们分别是1主键索引树非叶节点 2主键索引树叶子节点 3辅助键索引树非叶节点 4辅助键索引树叶子节点。这4种节点的Record格式有一些差异，但是它们都存储着Next指针指向下一个Record。



User Record在Page内以单链表的形式存在，最初数据是按照插入的先后顺序排列的，但是随着新数据的插入和旧数据的删除，数据物理顺序会变得混乱，但他们依然保持着逻辑上的先后顺序。



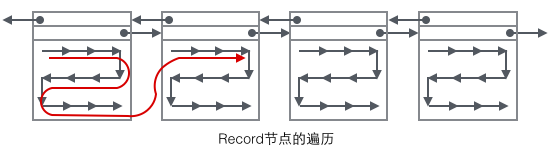
把User Record的组织形式和若干Page组合起来，就看到了稍微完整的形式



现在看下如何定位一个Record：

　　1 通过根节点开始遍历一个索引的B+树，通过各层非叶子节点最终到达一个Page，这个Page里存放的都是叶子节点。

　　2 在Page内从"Infimum"节点开始遍历单链表（这种遍历往往会被优化），如果找到该键则成功返回。如果记录到达了"supremum"，说明当前Page里没有合适的键，这时要借助Page的Next Page指针，跳转到下一个Page继续从"Infimum"开始逐个查找。



## 聚簇索引和非聚簇索引区别

聚簇索引是顺序结构与数据存储物理结构一致的一种索引，并且一个表的聚簇索引只能有唯一的一条；表数据按照索引的顺序来存储的，也就是说索引项的顺序与表中记录的物理顺序一致。对于聚集索引，叶子结点即存储了真实的数据行，不再有另外单独的数据页。

非聚簇索引记录的物理顺序与逻辑顺序没有必然的联系，与数据的存储物理结构没有关系；一个表对应的非聚簇索引可以有多条，根据不同列的约束可以建立不同要求的非聚簇索引；表数据存储顺序与索引顺序无关。对于非聚集索引，叶结点包含索引字段值及指向数据页数据行的逻辑指针

索引是通过二叉树的数据结构来描述的，我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而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

* 更新表数据

1、向表中插入新数据行  
    如果一张表没有聚集索引，那么它被称为“堆集”（Heap）。这样的表中的数据行没有特定的顺序，所有的新行将被添加到表的末尾位置。而建立了聚簇索引的数据表则不同：最简单的情况下，插入操作根据索引找到对应的数据页，然后通过挪动已有的记录为新数据腾出空间，最后插入数据。如果数据页已满，则需要拆分数据页，调整索引指针（且如果表还有非聚集索引，还需要更新这些索引指向新的数据页）。而类似于自增列为聚集索引的，数据库系统可能并不拆分数据页，而只是简单的新添数据页。

2、从表中删除数据行

    对删除数据行来说：删除行将导致其下方的数据行向上移动以填充删除记录造成的空白。如果删除的行是该数据页中的最后一行，那么该数据页将被回收，相应的索引页中的记录将被删除。对于数据的删除操作，可能导致索引页中仅有一条记录，这时，该记录可能会被移至邻近的索引页中，原索引页将被回收，即所谓的“索引合并”。

## 为什么使用B+树而不使用B树

<http://blog.csdn.net/fei33423/article/details/48469899>

<http://blog.csdn.net/cangchen/article/details/44818485>

1）B+树空间利用率更高，可减少I/O次数

     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。而因为B+树的内部节点只是作为索引使用，而不像B-树那样每个节点都需要存储硬盘指针。

**也就是说：**B+树中每个非叶节点没有指向某个关键字具体信息的指针，所以每一个节点可以存放更多的关键字数量，即一次性读入内存所需要查找的关键字也就越多，减少了I/O操作。

**e.g.**假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内   部结点需要2个盘快。而**B+**树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比**B+**树多一次盘块查找时间(在磁盘中就   是         盘片旋转的时间)。

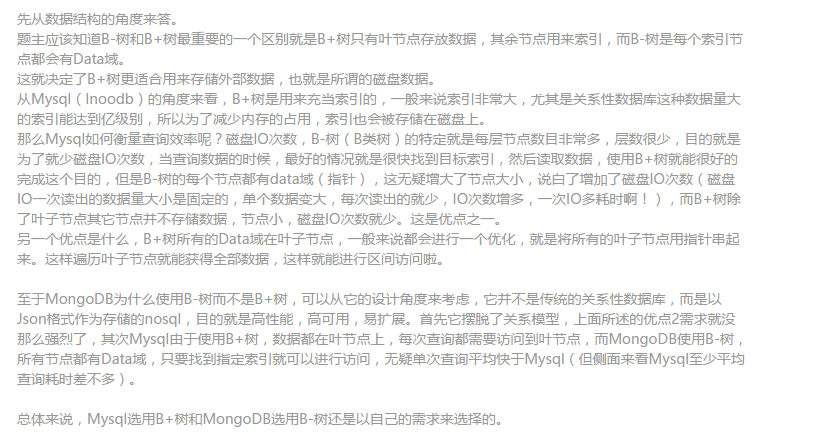
（2）增删文件（节点）时，效率更高，

**因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。**

（3）B+树的查询效率更加稳定，

因为B+树的每次查询过程中，都需要遍历从根节点到叶子节点的某条路径。所有关键字的查询路径长度相同，导致每一次查询的效率相当。

B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树需要遍历整棵树，效率太低。



### 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)。**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）。

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。

**综上所述，用B-Tree作为索引结构效率是非常高的。**

## 不能在where语句中使用聚合函数

问题：

Select \* from user where count(\*) >3 group by name;报错聚合函数不允许在这里使用

原因：sql语句的执行过程：from-where-group by-having-order by-select

聚合函数是针对结果集进行的，但是where条件并不是在查询出结果集之后运行，所以主函数放在where语句中，会出现错误。

而having不一样，having是针对结果集进行过滤的，所以可以把聚合函数放在having中，用having代替where。

Select \* from user group by name having count(\*)>3;

Having可以不和group by联合使用。如果没有group by，那么整个结果集就是一个分组。比如需要对现有的结果集做二次筛选时，就会直接使用having子句。

select \* from user where age>10 having max(age);

聚合函数也叫列函数，是对整列数据进行计算的，而where子句是对数据行进行过滤。因为聚合函数在对全列数据进行计算，因此使用它的前提是结果集必须确定。而在where中结果集并没有确定，还在筛选。

Where是对查询结果进行分组前，将不符合where条件的行去掉，即在分组前过滤数据，having是显示特定的组。

## 数据库事务的四个特性及含义

数据库事务transanction正确执行的四个基本要素。ACID,原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。  
**原子性**:整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。  
**一致性**:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。  
**隔离性**:隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。  
**持久性**:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

## 数据库的事务的隔离级别

低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。

### Read Uncommitted（读未提交）

A事务修改了数据库中的数据，但是还没他提交，B事务可以读取到这个修改之后的数据。但是由于某些原因，A事务进行回滚操作，之前修改的数据恢复成原来的数据了。这就造成了脏读。

很少实际中使用。

### Read Committed（读已提交）

B事务只能读取到A事务中已经修改之后的数据，这解决了脏读的问题。

但是可能出现不可重复读。在一个事务操作中，两次查询的结果不一致，这就是不可重复读。原因是在A事务操作过程，B事务修改了一些数据，并且已经提交，此时A事务可以读取到B事务中提交的数据，造成了前后读取数据的不一致。（一个交叉的事务）

或者对于同一个数据来说，被多个实例操作，同一事务的其他实例在该实例处理期间可能会有新的commit

### Repeatable Read(可重读)

在一个事务操作中，前后读取的数据是一样的，此时不会读取到另外一个事务已经提交过来的数据。解决了不可重复读的问题。

但是可能出现幻读。A事务在操作过程中查询了一个范围内的记录，此时B事务在该范围内新插入了一条记录，并且提交，A事务可能会读取到这个新插入的记录。

幻读注重的是原先不存在的数据，在这次查询时突然出现了。而可重复读强调的是对已经存在的数据进行修改，在下次查询时能不能看得到的问题。

Mysql的默认隔离级别。确保了同一个事务的实例在并发读取数据时，会看到同样的数据。

**InnoDB和Falcon存储引擎通过MVCC（多版本并发机制）解决了幻读问题。**

### **Serializable（可串行化）**

这是最高的隔离级别，通过强制事务进行排序，使之不可能相互冲突，从而解决了幻读问题。它是在每个读的数据行上加了共享锁。

但是可能导致大量的超时现象和锁竞争。

### 总结

读未提交：任何操作都不加锁。

读已提交(RC)：行锁。只能防止别的事务进行数据的修改和删除，因此解决了不可重复读的问题。读取的时候不加锁，写入 修改和删除数据时需要加锁。

可重复读(RR)：行锁防止别的事务进行数据的修改或者删除，GAP（间隙）锁防止别的事务新增数据，行锁和间隙锁结合形成了Next-Key锁共同解决了RR级别的不可重复读和幻读问题。

可串行化：**读加共享锁，写加排他锁，读写互斥。**并发能力差。适合于并发少同时要求数据及时可靠的场景。不可重复读重点在于update和delete，而幻读的重点在于insert。

如果使用锁机制来实现这两种隔离级别，在可重复读中，该sql第一次读取到数据后，就将这些数据加锁，其它事务无法修改这些数据，就可以实现可重复读了。但这种方法却无法锁住insert的数据，所以当事务A先前读取了数据，或者修改了全部数据，事务B还是可以insert数据提交，这时事务A就会发现莫名其妙多了一条之前没有的数据，这就是幻读，不能通过行锁来避免。需要Serializable隔离级别 ，**读用读锁，写用写锁，读锁和写锁互斥，这么做可以有效的避免幻读、不可重复读、脏读等问题，**但会极大的降低数据库的并发能力。

Mysql采用了基于乐观锁的MVCC来避免了不可重复读和幻读的问题。

### 案例分析

#### 读未提交

**修改隔离级别：**

set tx\_isolation='READ-UNCOMMITTED';

select @@tx\_isolation;

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

| @@tx\_isolation |

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

| READ-UNCOMMITTED |

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

A：启动一个事务

start transaction;

select \* from tx;

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

| id | num |

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

| 1 | 1 |

| 2 | 2 |

| 3 | 3 |

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

事务B：启动一个事务，并执行更新语句，但不提交

start transaction;

update tx set num=10 where id=1;

select \* from tx;

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

| id | num |

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

| 1 | 10 |

| 2 | 2 |

| 3 | 3 |

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

在事务A中查询：

select \* from tx;

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

| id | num |

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

**| 1 | 10 |**

| 2 | 2 |

| 3 | 3 |

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

可以看到在A中已经读取到了B事务中未提交的数据。

此时事务B进行回滚操作，再次在A中进行查询的时候，数据就会变成B没有提交的数据。

select \* from tx;

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

| id | num |

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

| 1 | 1 |

| 2 | 2 |

| 3 | 3 |

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

#### 读已提交

**修改隔离级别**

set tx\_isolation='read-committed';

select @@tx\_isolation;

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

| @@tx\_isolation |

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

| READ-COMMITTED |

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

**A中启动一个事务**

start transaction;

select \* from tx;

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

| id | num |

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

| 1 | 1 |

| 2 | 2 |

| 3 | 3 |

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

**事务B：也启动一个事务(那么两个事务交叉了) 在这事务中更新数据，且未提交**

start transaction;

update tx set num=10 where id=1;

select \* from tx;

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

| id | num |

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

| 1 | 10 |

| 2 | 2 |

| 3 | 3 |

**在事务A中查询，数据没变，没有脏读的出现。**

select \* from tx;

| id | num |

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

| 1 | 1 |

| 2 | 2 |

| 3 | 3 |

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

**B提交事务**

commit;

**再次在A中查询：**

select \* from tx; --------------->

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

| id | num |

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

| 1 | 10 |

| 2 | 2 |

| 3 | 3 |

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

**发现两次查询的结果不一致，这就是不可重复读。**

#### 可重复读

**修改隔离级别**  
set tx\_isolation='repeatable-read';

select @@tx\_isolation;

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

| @@tx\_isolation |

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

| REPEATABLE-READ |

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

**事务A：启动一个事务**

start transaction;

select \* from tx;

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

| id | num |

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

| 1 | 1 |

| 2 | 2 |

| 3 | 3 |

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

**事务B：开启一个新事务(那么这两个事务交叉了)在事务B中更新数据，并提交**

start transaction;

update tx set num=10 where id=1;

select \* from tx;

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

| id | num |

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

| 1 | 10 |

| 2 | 2 |

| 3 | 3 |

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

commit;

**在A中进程查询，读取不到更改之后的数据，说明没有不可重复读的问题。**

select \* from tx;

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

| id | num |

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

| 1 | 1 |

| 2 | 2 |

| 3 | 3 |

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

**事务A：只有当事务A也提交了，它才能够看到数据变化**

commit;

select \* from tx;

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

| id | num |

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

| 1 | 10 |

| 2 | 2 |

| 3 | 3 |

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

#### 可串行化

**修改隔离级别**  
set tx\_isolation='serializable';

select @@tx\_isolation;

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

| @@tx\_isolation |

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

| SERIALIZABLE |

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

**事务A：开启一个新事务**

start transaction;

**事务B：在A没有commit之前，这个交叉事务是不能更改数据的**

start transaction;

insert tx values('4','4');

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

update tx set num=10 where id=1;

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

### 总结



## 数据库的隔离级别以及对应的锁机制

幻读的问题应该在最高的隔离级别—可串行化中得以解决，但是在Mysql数据库中，在可重复读的隔离级别就解决了幻读的问题，**InnoDB和Falcon存储引擎通过MVCC（多版本并发机制）解决了幻读问题。**

### MVCC

Multi-Version Concurrency Control 并发多版本控制

MVCC的实现,是通过保存数据在某个时间点的快照来实现的.

InnoDB的MVCC是通过在每行记录后面保存2个隐藏的列来实现的,一列保存了行的创建时间,一列保存了行的过期时间(或删除时间).存储的不是实际时间，而是系统版本号。

每开始一个新的事务，系统版本号就会自动递增，事务开始时刻的系统版本号会作为事务版本号，用来和查询到的每行记录的版本号进行比较。

MVCC最大的作用是:  实现了非阻塞的读操作,写操作也只锁定了必要的行.

MYSQL的MVCC 只在 read committed  和 repeatable read  2个隔离级别下工作.

在MVCC的机制下,mysql InnoDB(**默认隔离级别也就是可重复读**)的增删改查变成了如下模式:

SELECT:    
1, InnoDB只查找版本早于当前事务版本的数据行(行的系统版本号小于等于事务的系统版本号) 这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改的。  
2, 行的删除版本要么未定义,要么大于当前事务版本号,这样可以确保事务读取到的行,在事务开始之前未被删除.  
只有符合上述两个条件的记录才能作为查询的结果

保存这两个额外的系统版本号，使大多数读操作可以不用加锁，这样设计使得读数据操作很简单，性能很好，并且保证只会读取到符合标准的行。不足之处，每行都需要额外的存储空间，需要做更多的检查和维护工作。

MVCC只会在可重复读和读已提交两个级别下工作。因为读未提交每次读取的总是最新的数据行，而不是符合当前事务版本的数据行，而可串行化会对所有要读取的行都加锁。  
INSERT:  
InnoDB 为新插入的每一行保存当前系统版本号做为行版本号。  
  
DELETE:   
INNODB 为删除的每一行保存当前系统版本号作为行删除标识  
  
UPDATE:  
InnoDB 为插入的每一行新记录,保存当前系统版本号作为行版本号,同时保存当前系统版本号到原来的行作为行删除标识.

我们且看，在RR级别中，通过MVCC机制，虽然让数据变得可重复读，但我们读到的数据可能是历史数据，是不及时的数据，不是数据库当前的数据！这在一些对于数据的时效特别敏感的业务中，就很可能出问题。

对于这种读取历史数据的方式，我们叫它快照读 (snapshot read)，而读取数据库当前版本数据的方式，叫当前读 (current read)。很显然，在MVCC中：

* 快照读：就是select
  + select \* from table ....;
* 当前读：特殊的读操作，插入/更新/删除操作，属于当前读，处理的都是当前的数据，需要加锁。
  + select \* from table where ? lock in share mode;
  + select \* from table where ? for update;
  + insert;
  + update ;
  + delete;

事务的隔离级别实际上都是定义了当前读的级别，MySQL为了减少锁处理（包括等待其它锁）的时间，提升并发能力，引入了快照读的概念，使得select不用加锁。而update、insert这些“当前读”，就需要另外的模块来解决了。

事务的隔离级别中虽然只定义了读数据的要求，实际上这也可以说是写数据的要求。上文的“读”，实际是讲的快照读；而这里说的“写”就是当前读了。  
为了解决当前读中的幻读问题，MySQL事务使用了Next-Key锁。

一致性读是通过MVCC为查询提供了一个基于时间点的快照。这个查询只能看到自己之前提交的数据，而在查询开始之后提交的数据是无法看到的。但是这个查询可以看到在自己开始之后的同一个事务产生的变化。

在默认隔离级别下RR下，同一事务的所有一致性读只会读取第一次查询时创建的快照。

http://www.cnblogs.com/metoy/p/5545580.html

### 不加锁的读（一致性非锁定读）

是InnoDB存储引擎下的读取数据的方式( read committed  和 repeatable read).

一致性非锁定读,我的理解是它的读取方式是把: 事务隔离级别,MVCC,InnoDB锁结合起来运用到实现 Mysql读的一种方式.

Mysql读取数据的方式是读MVCC下的快照数据。具体的说，读取mysql数据库时,如果读取的行正在执行DELETE,UPDATE等操作,这时,读取操作不会因此去等待行上的X锁释放,相反,InnoDB会读取行的一个快照数据.**这样利用MVCC,InnoDB实现了非阻塞读，极大的提高了数据库的并发性.**

**但在不同的事务隔离级别下读取的数据方式也不一样。**

(1).  在read committed隔离级别下:

一致性非锁定读总是读取被锁定行的最新一份快照数据.  产生了不可重复读的问题.

(2).  在repeatable read 事务隔离级别下:

 一致性非锁定读总是读取事务开始时的行数据版本.  解决不可重复读的问题

### 加锁的读（一致性锁定读）

1). select .... for update. 加X锁

2). select .... lock in share mode. 加S锁

### 原理

Innodb行级锁分为

Record lock:单行记录的锁

Gap lock:间隙锁，锁定的是一个范围，但不包括记录本身

Next-key lock:锁定一个范围并锁定记录本身。上面两种的结合。

上述锁定的对象指的是索引记录。如果innodb存储引擎在建立的时候没有设置任何一个索引，那么innodb会使用隐式的主键进行锁定。当查询的索引含有唯一的属性时，innodb存储引擎会对key-next lock进行优化，降级为record lock.

InnoDB存储引擎默认的事务隔离级别(repeatable read)下,采用的是 Next-Key Locking的方式来加锁.

read committed隔离级别下采用的是: Record Lock 的方式来加锁.

下面我们来看下 Next-key lock的具体实现:

默认存储引擎下, 比如表A 上的id字段有索引abc, 并且id有 3,8,12,20这几个值,那么该索引可能被Next-key locking区间为:

(负无穷,3)

[3,8)

[8,12),

[12,20),

[20,正无穷)

当事务T1锁定了 [8,12),[12,20)这2个区间时,当插入15时,上面的区间变成:

[8,12),[12,15),[15,20).

但查询索引含有唯一属性时,Next-Key Lock 降级为 Record Lock,仅锁住索引本身.

好,现在表A的id值变成了: 3,8,12,15,20

如果执行下列语句:

select \* from A where id>16 for update.

InnoDB会对(16,正无穷) 加锁,

但在 read committed的事务隔离级别下,因为采用Record Lock,只会锁定20这个值.

如果在此时另外一个事务T2,插入了22这个值,此时,  read committed 隔离级别下就会产生"幻读"的问题.

但在InnoDB默认存储引擎下的Next-key Lock 模式下,22是插入是会被阻塞的,直到事务T1提交后,释放X锁,才能提交22这值.这样,InnoDB就这样解决了幻读的问题.

### 总结

1, InnoDB用MVCC来实现非阻塞的读操作,不同隔离级别下,MVCC通过读取不同版本的数据来解决"不可重复读" 的问题.

2, InnoDB的默认隔离级别解决2个问题,"不可重复读" 和 "幻读", oracle需要在串行读中解决"幻读"问题. InnoDB的实现方式和一般隔离级别的定义不一致.

**不可重复读问题解决 是因为在读取数据时，只会读取早于当前事务版本号的数据行，如果期间其他事务提交了更改之后的数据，事务版本号比当前事务大，所以读取不到。**

3, InnoDB的默认隔离级别采用Next-key Lock(间隙锁) 来解决幻读问题. 而 read committed隔离级别采用Record锁,因此会产生"幻读"问题.

4, InnoDB的存储引擎不存在锁升级的问题(太多的行锁升级为表锁),来降低锁的开销. 因为不是根据记录来产生行锁的,根据页对锁进行管理.

## 乐观锁实现

乐观锁：假设不会发生并发冲突，只是在提交的时候检查是否违反了数据完整性。乐观锁不能解决脏读问题。

乐观锁一般通过数据版本实现。也就是为数据增加一个版本标识，可以为数据库表增加一个数字类型的version字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version增加1.当提交更新时，判断数据库表对应记录的当前的版本信息与第一次取出来的version值进行对比，如果数据库表当前的版本号和第一次取出来的version值是否相等，是的话则更新，否则则是过期数据。

## Mysql的锁机制

**为了保证数据的一致完整性，任何一个数据库都存在锁定机制。锁定机制的优劣直接影响到一个数据库系统的并发处理能力和性能。**

MyISAM/MEMORY/CSV存储引擎（非事务性存储引擎）采用的表级锁，BDB采用的是页面锁，也支持表级锁，InnoDB存储引擎既支持行级锁，也支持表级锁，默认情况下采用行级锁。

**表级锁**：开销小，加锁块；不会出现死锁，锁定粒度大，发生锁冲突的概率最高，并发度最低。

**因为一次性获得所需的全部锁，要么全部满足要么全部等待。故不会有死锁。**

**行级锁**：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发性也最高。

锁是逐步获取的，就造成了死锁的可能。

Mysql行级锁锁的是索引，而不是记录。

索引分为主键索引和非主键索引两种：

如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；

如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。

在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，即所谓的next-key locking。

**页面锁**：开销和加锁界于表锁和行锁之间，会出现死锁；锁定粒度界与表锁和行锁之间，并发一般。

### 行级锁

<http://blog.csdn.net/alexdamiao/article/details/52049993>

<http://blog.csdn.net/zwan0518/article/details/17316931>

行级锁是锁定粒度最小的，所发生的锁定资源争用的概率也最小，能够给予应用程序尽可能大的并发处理能力而提高一些需要高并发应用系统的整体性能。

但是由于锁定粒度比较小，所以每次获取锁和释放锁消耗的资源也更多，也最容易发生死锁。

行级锁定不是MySQL自己实现的锁定方式，而是由其他存储引擎自己所实现的，如广为大家所知的Innodb存储引擎，以及MySQL的分布式存储引擎NDBCluster等都是实现了行级锁定。

Innodb的行级锁定同样分为两种类型，**共享锁**和**排他锁**，而在锁定机制的实现过程中为了让行级锁定和表级锁定共存，Innodb也同样使用了意向锁（表级锁定）的概念，也就有了**意向共享锁**和**意向排他锁**这两种。

**共享锁**，允许多个线程同时获取一个锁，一个锁可以同时被多个线程拥有。

**排它锁**，也称作独占锁，一个锁在某一时刻只能被一个线程占有，其它线程必须等待锁被释放之后才可能获取到锁。

**共享锁**(s)：允许一个事务去读一行，阻止其他事务获取相同数据集的排他锁。

**排他锁**(x)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

例：假设有两个事务t1和t2  
如果事务t1获取了一个元组的共享锁，事务t2还可以立即获取这个元组的共享锁，但不能立即获取这个元组的排它锁（必须等到t1释放共享锁之后）。  
如果事务t1获取了一个元组的排它锁，事务t2不能立即获取这个元组的排共享锁，也不能立即获取这个元组的排它锁（必须等到t1释放排它锁之后）。

**意向锁**：意向锁是一种表锁，锁定的粒度是整张表，分为意向共享锁(IS)和意向排它锁(IX)两类。意向共享锁表示一个事务有意对数据上共享锁或者排它锁。“有意”这两个字表达的意思比较微妙，说的明白点就是指事务想干这个事但还没真去干。举例说明下意向共享锁，比如一个事务t执行了这样一个语句：select \* from table lock in share model ，如果这个语句执行成功，就对表table上了一个意向共享锁。lock in share model就是说事务t1在接下来要执行的语句中要获取S锁。如果t1的select \* from table lock in share model执行成功，那么接下来t1应该可以畅通无阻的去执行只需要共享锁的语句了。意向排它锁的含义同理可知，上例中要获取意向排它锁，可以使用select \* from table for update 。

排它锁、意向共享锁、意向排它锁相互之前都是有兼容/互斥关系的，可以用一个兼容性矩阵表示(y表示兼容，n表示不兼容):  
    X    S    IX    IS  
X  n     n    n     n  
S   n     y    n     y  
IX n     n    y     y  
IS n     y    y     y

意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁

意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

意向锁是InnoDB自动加的，不需要用户干预。

对于UPDATE、DELETE、INSERT语句，Innodb会自动给涉及的数据集加排他锁（X）；对于普通SELECT语句，Innodb不会加任何锁。

显示添加锁

共享锁（S）： SELECT \* FROM table\_name WHERE .... LOCK IN SHARE MODE

排他锁（X）：  SELECT \* FROM table\_name WHERE .... FOR UPDATE.

使用select ... in share mode获取共享锁，主要用在需要数据依存关系时，确认某行记录是否存在，并确保没有人对这个记录进行update或者delete。

当对某个资源加锁时，如果   
- 有共享锁，可以再加一个共享锁，不过不能加排他锁。   
- 有排他锁，就在表上添加意向共享锁或意向排他锁。

意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在。所以，可以说Innodb的锁定模式实际上可以分为四种：共享锁（S），排他锁（X），意向共享锁（IS）和意向排他锁（IX）

Innodb与Oracle锁实现机制的区别

* Oracle锁定数据是通过需要锁定的某行记录所在的物理block上的事务槽上表级锁定信息（通过在数据块中，对相应数据行加锁来实现）
* Innodb的锁定则是通过在指向数据记录的第一个索引键之前和最后一个索引键之后的空域空间上标记锁定信息而实现的。（**通过给索引上的索引项加锁实现的**）

**InnoDB这种行锁意味着 只有通过索引条件检索数据时，InnoDB才使用行锁，否则InnoDB将使用表锁。**

**Innodb的这种锁定实现方式被称为“NEXT-KEYlocking”（间隙锁），因为Query执行过程中通过过范围查找的话，他会锁定整个范围内所有的索引键值，即使这个键值并不存在。**

**间隙锁锁定的是一个范围。是为了防止幻读的出现。**

除了间隙锁给Innodb带来性能的负面影响之外，通过索引实现锁定的方式还存在其他几个较大的**性能隐患**：

* **当Query无法利用索引的时候，会放弃行级别锁定而改用表级别的锁定**
* **当Quuery使用的索引并不包含所有过滤条件的时候，间隙锁会锁定不包含的记录，而不是具体的索引键**
* **当Query在使用索引定位数据的时候，如果使用的索引键一样但访问的数据行不同的时候（索引只是过滤条件的一部分），一样会被锁定**

**MySQL InnoDB支持三种行锁定方式：**

**行锁**（Record Lock）：锁直接加在索引记录上面。只有通过索引条件检索数据才会使用行锁。否则将会使用表锁。针对的是索引进行加锁，即便不是同一条记录，但是如果使用相同的索引键，就会出现锁冲突。

**间隙锁（**Gap Lock）：锁加在不存在的空闲空间（记录），可以是两个索引记录之间，也可能是第一个索引记录之前或最后一个索引之后的空间。InnoDB使用间隙锁的目的：一是为了防止幻读，二是为了满足其恢复和复制的需要。

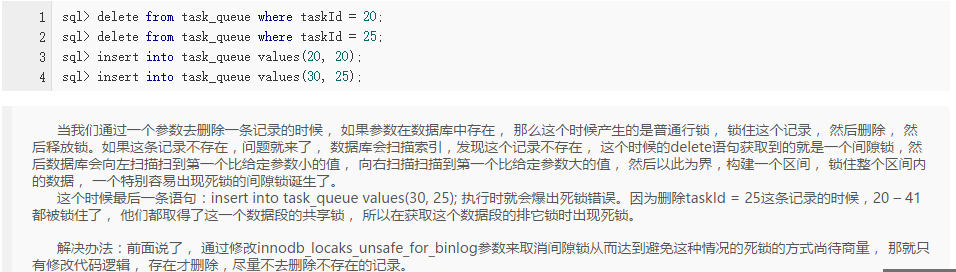
Next-Key Lock：行锁与间隙锁组合起来用就叫做Next-Key Lock。

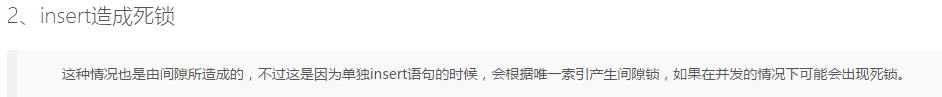
默认情况下，InnoDB工作在可重复读隔离级别下，并且以Next-Key Lock的方式对数据行进行加锁，这样可以有效防止幻读的发生。Next-Key Lock是行锁与间隙锁的组合，这样，当InnoDB扫描索引记录的时候，会首先对选中的索引记录加上行锁（Record Lock），再对索引记录两边的间隙（向左扫描扫到第一个比给定参数小的值， 向右扫描扫描到第一个比给定参数大的值， 然后以此为界，构建一个区间）加上间隙锁（Gap Lock）。如果一个间隙被事务T1加了锁，其它事务是不能在这个间隙插入记录的。

**间隙锁在InnoDB的唯一作用就是防止其它事务的插入操作，以此来达到防止幻读的发生，所以间隙锁不分什么共享锁与排它锁。**

**如果InnoDB扫描的是一个主键、或是一个唯一索引的话，那InnoDB只会采用行锁方式来加锁，而不会使用Next-Key Lock的方式，也就是说不会对索引之间的间隙加锁**

要禁止间隙锁的话，可以把隔离级别降为***读已提交***，或者开启参数***innodb\_locks\_unsafe\_for\_binlog***。





实例一：

建立t1表如下：

**[sql]** [view plain](http://blog.csdn.net/hsd2012/article/details/51112009) [copy](http://blog.csdn.net/hsd2012/article/details/51112009)

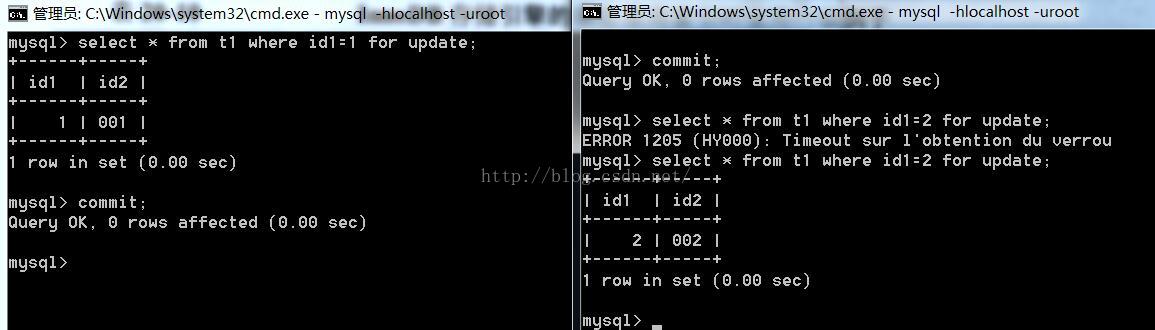
[在CODE上查看代码片](https://code.csdn.net/snippets/1641418)

1. **CREATE** **TABLE** `t1` (
2. `id1` **int**(5) **DEFAULT** NULL,
3. `id2` **int**(3) unsigned zerofill NOT NULL **DEFAULT** '000'
4. ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8

**[sql]** [view plain](http://blog.csdn.net/hsd2012/article/details/51112009) [copy](http://blog.csdn.net/hsd2012/article/details/51112009)

[在CODE上查看代码片](https://code.csdn.net/snippets/1641418)

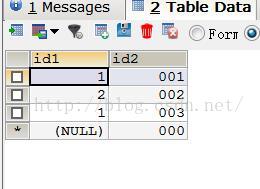
1. <span style="font-size:18px;">**insert** **into** t1 valuses(1,1),(2,2);</span>



因为没有创建索引，当给第一个会话添加索引时候，其实添加的是表索引，而非行索引，因为第二会话在查询其他信息时候，一直处于等待状态，最后超时，直到第一个会话事务提交后，方可查询。（需要先设置 set autocommit=0）

实例二：

修改上面t1表中数据，数据如下

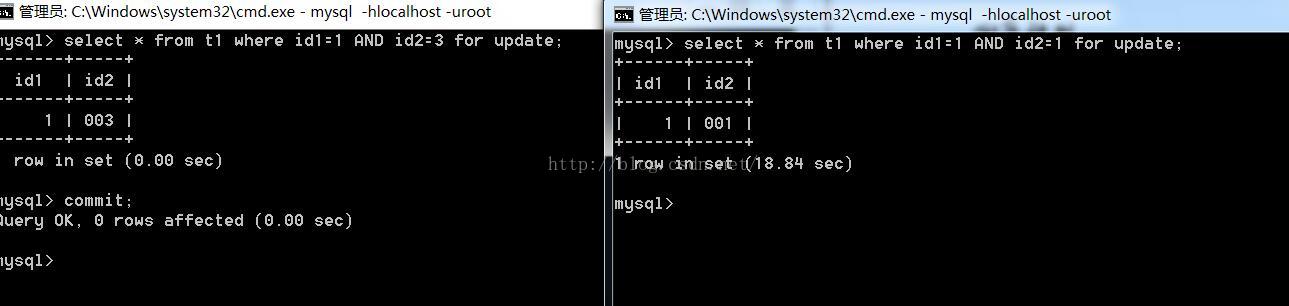


给id1添加索引ALTER TABLE t1 ADD INDEX id1(id1);



有此可以看出此时，mysql使用的是行索引。

但是还有一个需要我们注意



很明显两个会话查询的不是同一行记录，为什么会话2仍然需要等待会话1提交之后才能查询呢？还是因为**Mysql行锁是针对索引加的锁，不是针对记录加的锁，索引虽然访问不同的记录，但是他们的索引相同，是会出现冲突的，在设计数据库时候需要注意这一点。上面只有将字段id2，也添加上索引才能解决冲突问题。**这也是mysql效率低的一个原因。

### 表级锁

表级锁是最大颗粒度的锁定机制。实现逻辑简单，带来系统负面影响小，所以获取锁和释放锁的速度很快。由于表级锁一次会将整个表锁定，因此可以很好的避免死锁问题。

由于锁定颗粒度比较大，出现锁定资源争用的概率比较高，并发度较低。

表级锁定分为**读锁定和写锁定**。

在MySQL中，主要通过四个队列来维护这两种锁定：两个存放当前正在锁定中的读和写锁定信息，另外两个存放等待中的读写锁定信息，如下：

* **Current read-lock queue (lock->read)** 持有读锁的所有线程
* **Pending read-lock queue (lock->read\_wait)** 等待读锁的所有线程
* **Current write-lock queue (lock->write)** 持有写锁的所有线程
* **Pending write-lock queue (lock->write\_wait)** 等待写锁的所有线程

当前持有读锁的所有线程的相关信息都能够在Currentread-lockqueue中找到，队列中的信息按照获取到锁的时间依序存放。而正在等待锁定资源的信息则存放在Pendingread-lockqueue里面，另外两个存放写锁信息的队列也按照上面相同规则来存放信息。

**读锁是共享锁，写锁是排他锁。**

**读操作时，不会阻塞其他用户对同一个表的读请求，但是会阻塞对同一个表的写请求。**

**写操作时，会阻塞其他用户对同一个表的读和写操作。**

**表的读写操作之间以及写操作之间是串行的。**

**Myisam在执行查询之间会自动执行表的加锁和解锁操作。但是有时候也需要显示加锁。**

**比如：检索某一个时刻t1，t2表中数据数量。**

**常用代码如下：**

**select count(t1.id1) as 'sum' from t1;**

**select count(t2.id1) as 'sum' from t2;**

**其实这是不正确的，很有可能当你在检索t1的那个时间点，t2的数据已经发生了变化，也就是说你检查出的t1和t2数据结果不是在同一个时间点上。**

1. lock **table** t1 **read**, t2 **read**;
2. **select** count(t1.id1) **as** 'sum' **from** t1;
3. **select** count(t2.id1) **as** 'sum' **from** t2;
4. unlock tables;

**或者**

1. **SELECT**
2. COUNT(t1.`id1`) **AS** dadasum,'t1' **AS** tablename
3. **FROM**
4. t1
5. **UNION**
6. ALL
7. **SELECT**
8. COUNT(t2.`id1`)**AS** dadasum ,'t2' **AS** tablename
9. **FROM**
10. t2 ;

**注意：**

**1.在锁定表时候，如果加上关键字local，满足myISAM表的并发插入问题。eg： lock table t3 read local;**

**2.使用locak tables 给表加锁时候，必须同时给所有涉及到的表加锁，因为加锁之后，当前会话，就不能操作没有加锁的表**

**读锁和写锁是互斥的，如果读写两个进程同时请求一张表，Mysql会使写进程先获得锁。即使读请求先到达锁等待队列，写锁后到达，写锁也会先执行。因为写的优先级更高。这也是Myisam不适合含有大量更新操作和查询操作应用的原因。**

**解决方案：**

**1)通过指定启动参数low-priority-updates,使MyISAM引擎默认给与读请求优先的权限**

**2）通过执行set low\_PRIORITY\_UPDATES=1，降低更新请求的优先级。**

**3）指定INSERT、UPDATE、DELETE语句的LOW\_PRIORITY属性。**

#### 读锁定

一个新的客户端请求在申请获取读锁定资源的时候，需要满足两个条件：

1. 资源没有被**写锁定**
2. **写锁定**等待队列（Pendingwrite-lockqueue）中没有更高优先级的写锁定等待

如果满足了上面两个条件之后，该请求会被立即通过，并将相关的信息存入Currentread-lockqueue中，否则会被迫进入等待队列Pendingread-lockqueue中等待资源的释放。

#### 写锁定

一个新的客户端请求在申请获取写锁定资源的时候，被申请资源需要满足两个条件：

1. 没被写锁定Currentwrite-lockqueue
2. 没被写锁定等待(Pendingwrite-lockqueue)
3. 没被读锁定Currentread-lockqueue

Myisam在执行查询语句之前会自动给涉及的所有表加读锁，在执行更新操作前会自动给表加写锁。这个过程并不需要用户参与。也就是说可以不显示的进行加锁。

当给myisam表显示加锁的时候，是为了在一定程度上模拟事务操作。比如两张表：订单表和订单详细表，需要比较这两个表的总金额是否一致。此时需要显示加锁。

1. Lock tables orders read local, order\_detail read local;
2. Select sum(total) from orders;
3. Select sum(subtotal) from order\_detail;
4. Unlock tables;

LOCK TABLES时加了“local”选项，其作用就是在满足MyISAM表并发插入条件的情况下，允许其他用户在表尾并发插入记录，有关MyISAM表的并发插入问题。

在用LOCK TABLES给表显式加表锁时，必须同时取得所有涉及到表的锁，并且MySQL不支持锁升级。也就是说，在执行LOCK TABLES后，只能访问显式加锁的这些表，不能访问未加锁的表；同时，如果加的是读锁，那么只能执行查询操作，而不能执行更新操作。其实，在自动加锁的 情况下也基本如此，MyISAM总是一次获得SQL语句所需要的全部锁。这也正是MyISAM表不会出现死锁（Deadlock Free）的原因。

### 页面锁

锁定颗粒度介于行级锁和表级锁之间，所以获取锁定资源所需要的开销和提供的并发处理能力也是在二者之间。页面锁和行级锁一样，会发生死锁。

### Myisam的并发插入

myISAM存储引擎有一个系统变量，concurrent\_insert，专门用来控制并发插入行为的，值可以为0，1，2.

concurrent\_insert为0时候，不允许插入(Never)

concurrent\_insert为1时候，如果mysql没有空洞（中间没有被删除的行），myISAM运行一个进程读表的时候，另一个进程从表尾插入记录，这也是mysql默认设置。(Auto)

concurrent\_insert为2时候，无论MyISAM表中有没有空洞，都允许在表尾并行的插入。(Always)

对于Local Table语句，Read Local和Read语句的区别：当加锁的时候，read local允许插入语句和并发插入语句的执行。

案例:

Lock table test read local;

同一会话内的插入或更新操作会失败

Insert into test values(,,);

Update test set name=’’ where id=’’;

在其他的会话2中：

允许插入操作:insert into test values(90,,);

但是此时查询会话2的新插入的行，会查询到空集

Select \* from test where id=90;

此时查看隔离级别:show variables like ‘%isolation%’;为可重复读

Unlock tables;

在会话2中，更新操作会等待：update test set name=’’ where id=;

直至会话1执行unlock tables,会话2才结束等待。

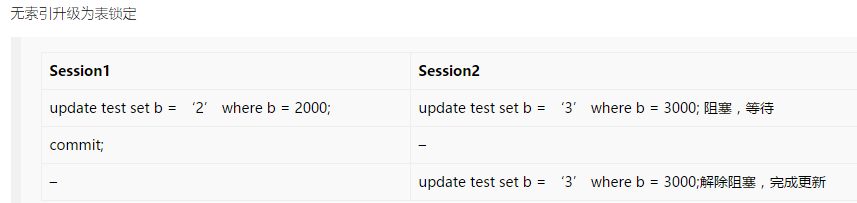
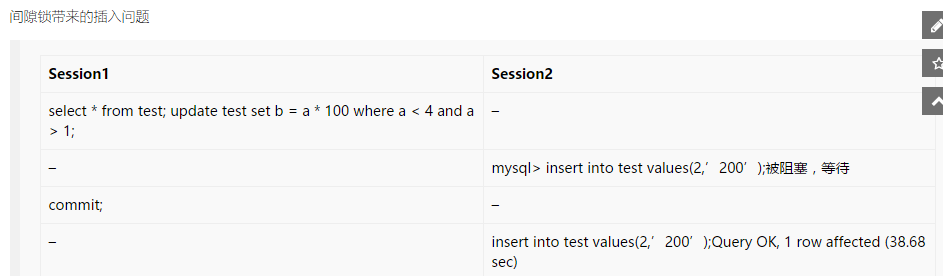
此时可以查询到会话2中插入的 记录。

### Innodb事务隔离级别下的锁定及死锁

在Innodb的事务管理和锁定机制中，有专门检测死锁的机制，会在系统中产生死锁之后的很短时间内就检测到该死锁的存在。

当Innodb检测到系统中产生了死锁之后，Innodb会通过相应的判断来选这产生死锁的两个事务中较小的事务来回滚，而让另外一个较大的事务成功完成。

但是有一点需要注意的就是，当产生死锁的场景中涉及到不止Innodb存储引擎的时候，Innodb是没办法检测到该死锁的，这时候就只能通过**锁定超时限制**来解决该死锁了。



### 合理利用锁机制优化mysql

#### MyISAM表锁优化建议

在优化MyISAM存储引擎锁定问题的时候，最关键的就是如何让其提高并发度。由于锁定级别是不可能改变的了，所以我们首先需要尽可能让**锁定的时间变短**，然后就是让可能并发进行的操作尽可能的并发。

1 缩短锁定时间

尽量**减少大Query**，将复杂Query分拆成小的Query分布进行；   
尽可能的建立足够**高效的索引**，让数据检索更迅速；   
尽量让MyISAM存储引擎的表只存放必要的信息，控制字段类型；   
利用合适的机会优化MyISAM表数据文件；

2分离能并行的操作

可能有些人会认为在MyISAM存储引擎的表上读写锁就只能是完全的串行化，没办法再并行了。大家不要忘记了，MyISAM的存储引擎还有一个非常有用的特性，那就是ConcurrentInsert（并发插入）的特性。   
Concurrent\_insert=2，无论MyISAM存储引擎的表数据文件的中间部分是否存在因为删除数据而留下的空闲空间，都允许在数据文件尾部进行   
ConcurrentInsert;concurrent\_insert=1，当MyISAM存储引擎表数据文件中间不存在空闲空间的时候，可以从文件尾部进行ConcurrentInsert;   
concurrent\_insert=0，无论MyISAM存储引擎的表数据文件的中间部分是否存在因为删除数据而留下的空闲空间，都不允许ConcurrentInsert。

3合理利用读写优先级

表级锁定对于读和写是有不同优先级设定的，默认情况下是写优先级要大于读优先级。所以，如果我们可以根据各自系统环境的差异决定读与写的优先级。如果我们的系统是一个以读为主，而且要优先保证查询性能的话，我们可以通过设置系统参数选项low\_priority\_updates=1，将写的优先级设置为比读的优先级低，即可让告诉MySQL尽量先处理读请求。

#### Innodb行锁优化建议

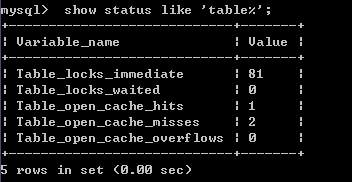
* 尽可能让所有的数据**检索都通过索引**来完成，从而避免Innodb因为无法**通过索引键加锁**而升级为表级锁定；
* 合理设计索引，让Innodb在索引键上面加锁尽可能准确，尽可能的**缩小锁定范围**，避免造成不必要的锁定而影响其他Query的执行；
* 尽可能减少基于范围的数据检索过滤条件，**避免间隙锁**带来的负面影响而锁定了不该锁定的记录；
* 尽量**控制事务的大小**，减少锁定的资源量和锁定时间长度；
* 在业务环境允许的情况下，尽量**使用较低级别的事务隔离**，以减少MySQL因为实现事务隔离级别所带来的附加成本；

### 减少死锁的方法

a. 类似业务模块中，尽可能按照相同的访问顺序来访问，防止产生死锁；   
b. 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；   
c. 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

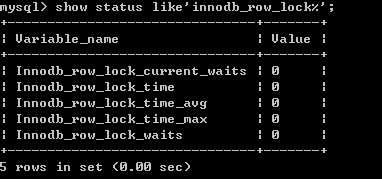
### 信息查询

系统锁定争用情况查询对于两种锁定级别，MySQL内部有两组专门的状态变量记录系统内部锁资源争用情况   
MySQL 实现的表级锁定的争用状态变量：



Table\_locks\_immediate：产生表级锁定的次数；   
Table\_locks\_waited：出现表级锁定争用而发生等待的次数；

对于Innodb所使用的行级锁定，系统中是通过另外一组更为详细的状态变量来记录的，如下：



Innodb 的行级锁定状态变量不仅记录了锁定等待次数，还记录了锁定总时长，每次平均时长，以及最大时长，此外还有一个非累积状态量显示了当前正在等待锁定的等待数量。对各个状态量的说明如下：

* Innodb\_row\_lock\_current\_waits：当前正在等待锁定的数量；
* Innodb\_row\_lock\_time：从系统启动到现在锁定总时间长度；
* Innodb\_row\_lock\_time\_avg：每次等待所花平均时间；
* Innodb\_row\_lock\_time\_max：从系统启动到现在等待最常的一次所花的时间；
* Innodb\_row\_lock\_waits：系统启动后到现在总共等待的次数；

对于这5个状态变量，比较重要的主要是**Innodb\_row\_lock\_time\_avg**（等待平均时长），**Innodb\_row\_lock\_waits**（等待总次数）以及**Innodb\_row\_lock\_time**（等待总时长）这三项。尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

此外，Innodb除了提供这五个系统状态变量之外，还提供的其他更为丰富的即时状态信息供我们分析使用。可以通过如下方法查看

1.通过创建InnodbMonitor表来打开Innodb的monitor功能：   
2.然后通过使用“show engine innodb status；”查看细节信息

## Lock in share mode|for update

SELECT ... LOCK IN SHARE MODE在查询到的行上设置共享锁。其他会话可以读取到这些行，但是不能修改这些行，直到提交事务后。  
如果将LOCK IN SHARE MODE用在支持页锁或行锁的存储引擎上，查询所覆盖的行会被加上共享锁，共享锁允许其他事务读，但是不允许其他事务更新或删除这些行。

LOCK for update加的是排他锁，不允许其他会话再次进行加锁。

## 为什么要有意向锁

意向锁（IX和IS）是一个表级锁，不会和行级锁发生冲突，只会和表级锁的X和S发生冲突。

Mysql中的表级锁有读锁和写锁，用读锁锁表时，会阻塞其他事务修改表的数据，但是允许共同去读取数据；使用写锁锁表时，会阻塞其他事务的读写操作。读锁是共享锁，写锁是排他锁。

Innodb支持行级锁，分为共享锁和排它锁。共享锁是一个事务对一行的共享只读锁；排它锁是一个事务对一行的排他读写锁。

这两中类型的锁共存的问题

事务A锁住了表中的**一行**，让这一行只能读，不能写。

之后，事务B申请**整个表**的写锁。

如果事务B申请成功，那么理论上它就能修改表中的任意一行，这与A持有的行锁是冲突的。

数据库需要避免这种冲突，就是说要让B的申请被阻塞，直到A释放了行锁。

数据库要怎么判断这个冲突呢？

step1：判断表是否已被其他事务用表锁锁表  
step2：判断表中的每一行是否已被行锁锁住。

注意step2，这样的判断方法效率实在不高，因为需要遍历整个表。  
于是就有了意向锁。

在意向锁存在的情况下，事务A必须先申请表的意向共享锁，成功后再申请一行的行锁。

在意向锁存在的情况下，上面的判断可以改成

step1：不变  
step2：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务B申请表的写锁会被阻塞。

意向锁是在添加行锁之前添加的。

当需要向一个表添加表级锁的时候，如果没有意向锁，则需要遍历整个表判断是否有行锁的存在，以免发生冲突，但是有了意向锁，只需要判断该意向锁与即将添加的表级锁是否兼容即可。因为意向锁的存在代表了有行级锁的存在或者即将有行级锁的存在，无需遍历整个表，效率大大提高了。

注意：意向锁的动作是数据库完成的，数据库会自动在申请行级锁之前添加意向锁。

## 视图的作用，视图可以更改么？

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；

数据库中只存放了视图的定义并没有存放视图的数据，数据还是存放在原来的表中。

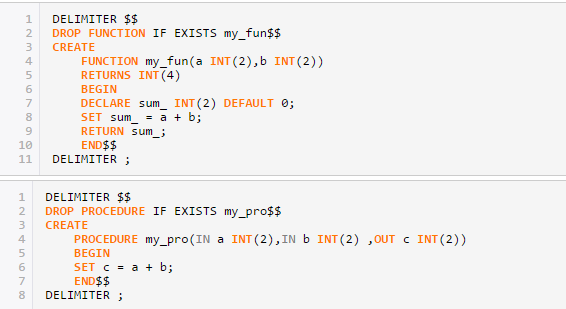
视图不需要实际上的物理存储，视图实际上是由预定义的查询形式的表所组成的。

使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。  
视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by 则对视图再次order by将被覆盖。  
创建视图：create view XXX as XXXXXXXXXXXXXX;  
对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

## Mysql函数

函数必须有指定的返回值，且参数默认为IN类型，而存储过程可以没有返回值，参数可以是IN OUT INOUT类型。

调用方式 select my\_fun(); call my\_proc();



## Mysql存储过程

Mysql储存过程是一组为了完成特定功能的SQL语句集，经过编译之后存储在数据库中，当需要使用该组SQL语句时用户只需要通过指定储存过程的名字并给定参数就可以调用执行。

优点：

1 存储过程可以实现较快的执行速度

如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在**每次运行时都要进行编译和优化**，速度相对要慢一些。  
心得：**编译优化，快！**

2存储过程可以用流控制语句编写，**有很强的灵活性，可以完成复杂的判断和较复杂的运算**。  
心得：**功能强大，逻辑强大**

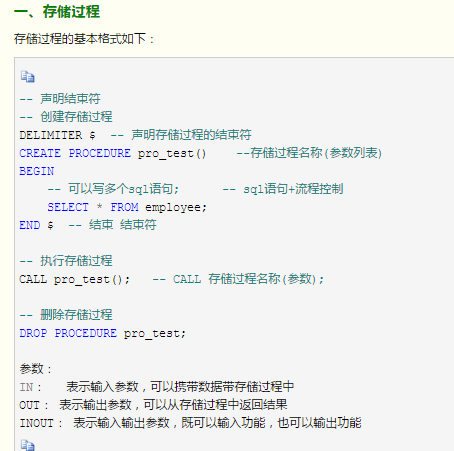
3 存储过程允许标准组件式编程。  
存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。  
心得：封装与抽象，简单调用

4 安全性比较高  
存储过程可以使用权限控制，而参数化的存储过程可以防止sql注入。  
心得：**限制与安全**

5减少网络流量  
针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织程存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，存储过程是在服务器上执行的，所有的数据访问都是在服务器内部进行，不需要传输数据到其他终端。从而大大减少了网络流量并降低了网络负载。  
心得：**减少网络流量（封装的好）**

缺点：

1 不能跨库移植，比如之前在mysql中的存储过程，如果要移植到oracle数据中，那么所有存储过程都要重写。

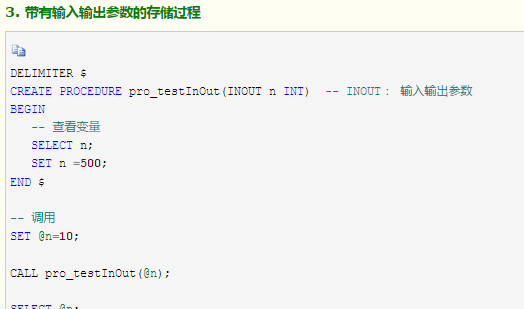
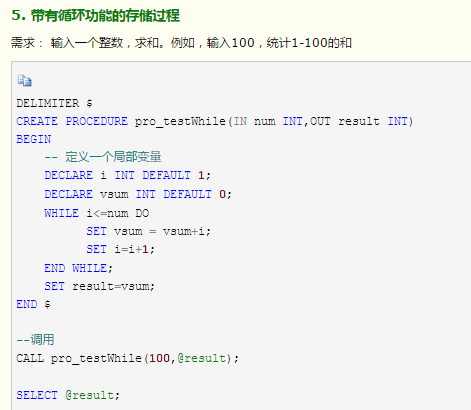
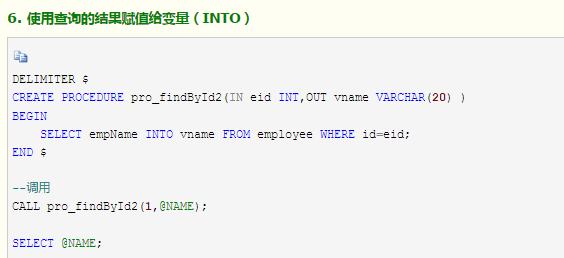
 

定义一个会话变量name, 使用name会话变量接收存储过程的返回值

CALL pro\_testOut(@NAME);

查看变量值

SELECT @NAME;

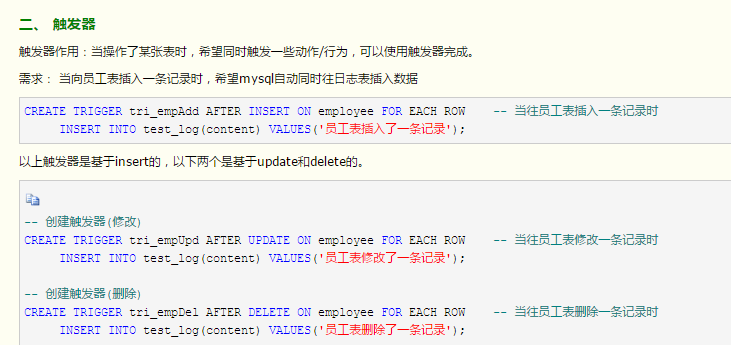
 MySQL默认将分号，即“;”作为语句的分隔符。如果是这样的话，则一个存储过程将很难正常创建，因为它的BEGIN和END之间可以是任意数量的SQL语句，而每条SQL语句都以分号结束。所以，在创建存储过程时，必须重新定义分隔符。在上述用例中，均将“$”作为新的分隔符，注意，在创建完存储过程后，必须重新设置回原来的分隔符“;”。

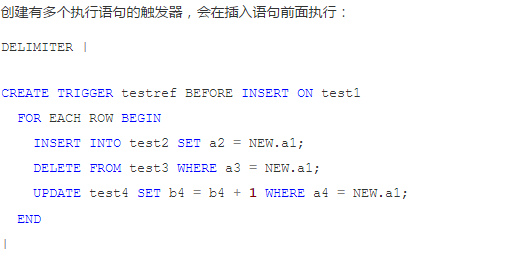
## Mysql触发器

触发器是一个特殊的存储过程，不同的是存储过程要用CALL来调用，而触发器不需要使用CALL

也不需要手工启动，只要当一个预定义的事件发生的时候，就会被MYSQL自动调用







## Mysql临时表

临时表是建立在系统临时文件夹中的表，如果使用得当，完全可以像普通表一样进行各种操作。 临时表的数据和表结构都储存在内存之中，退出时，其所占的空间会自动被释放。



临时表容量 tmp\_table\_size=1024M

**应用**：当工作在非常大的表上时，你可能偶尔需要运行很多查询获得一个大量数据的小的子集，不是对整个表运行这些查询，而是让MySQL每次找出所需的少数记录，将记录选择到一个临时表可能更快些，然后多这些表运行查询。

可以把一些经常访问的数据放到临时表中，这样访问时会快一些,因为数据是在服务器内存中,另外每次查询的时候，数据库都需要生成一些临时数据在临时表里

**注意**：

(1)临时表只在当前连接可见，当这个连接关闭的时候，会自动drop。这就意味着你可以在两个不同的连接里使用相同的临时表名，并且相互不会 冲突，或者使用 已经存在的表，但不是临时表的表名。(当这个临时表存在的时候，存在的表被隐藏了，如果临时表被drop，存在的表就可见了)。  
(2) 临时表只能用在 memory,myisam,merge,或者innodb引擎。  
3）临时表不支持mysql cluster(簇)。  
4）在同一个query语句中，你只能查找一次临时表。例如：下面的就不可用  
　　mysql> SELECT \* FROM temp\_table, temp\_table AS t2;  
　　ERROR 1137: Can't reopen table: 'temp\_table'  
　　如果在一个存储函数里，你用不同的别名查找一个临时表多次，或者在这个存储函数里用不同的语句查找，这个错误都会发生。  
5）show tables 语句不会列举临时表，但是会列出内存表。  
6）你不能用rename来重命名一个临时表。但是，你可以alter table代替：  
　　mysql>ALTER TABLE orig\_name RENAME new\_name;

## Mysql内存表

内存表:表结构建在磁盘里，数据在内存里 ，当停止服务后，表中的数据丢失，而表的结构不会丢失。内存表也可以被看作是临时表的一种。

CREATE TABLE tmp\_table (  
name VARCHAR(10) NOT NULL,         
value INTEGER NOT NULL   
)   ENGINE=MEMORY



应用：

内存表使用哈希散列索引把数据保存在内存中，因此具有极快的速度，适合缓存中小型数据库。  
1、内存表对所有用户的连接是可见的，这使得它非常适合做缓存。  
2、一旦服务器重启，内存表数据丢失，但是内存表结构仍然存在，因为内存表结构是存放在实际数据库路径下的，不会自动删除。重启之后，内存将被清空，这时候对内存表的查询结果都是空的。  
3、如果内存表是复制的某数据表（创建普通用户表这些也会丢失），则复制之后所有主键、索引、自增等格式将不复存在，需要重新添加主键和索引，如果需要的话。  
4、对于重启造成的数据丢失，有以下的解决办法：  
　a、在任何查询之前，执行一次简单的查询，判断内存表表是否存在数据，如果不存在，则把数据重新写入，或者DROP表重新复制某张表。这需要多做一次查询。不过可以写成include文件，在需要用该内存表表的页面随时调用，比较方便。  
　b、对于需要该内存表表的页面，在该页面第一次且仅在第一次查询该表时，对数据集结果进行判断，如果结果为空，则需要重新写入数据。这样可以节省一次查询。  
　c、更好的办法是在mysql每次重新启动时自动写入数据到内存表，但是需要配置服务器，过程比较复杂，通用性受到限制。

注意：内存表可以通过max\_heap\_table\_size = 2048M来加大使用的内存

内存表必须使用memory存储引擎

内存表不允许使用xxxTEXT和xxxBLOB数据类型；只允许使用=和<=>操作符来搜索记录（不允 许& amp; lt;、>、<=或>=）；mysql4.1版本之前不支持auto\_increment；只允许对非空数据列进行索引（not null）。  
注：操作符 “<=>” 说明：NULL-safe equal.这个操作符和“=”操作符执行相同的比较操作，不过在两个操作码均为NULL时，其所得值为1而不为NULL，而当一个操作码为NULL时，其所得值为0而不为NULL。

## drop,delete与truncate的区别

drop直接删掉表 truncate删除表中数据，再插入时自增长id又从1开始 delete删除表中数据，可以加where字句。

（1） DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对TABLE；DELETE可以是table和view

（5） TRUNCATE 和DELETE只删除数据，而DROP则删除整个表（结构和数据）。

（6） truncate与不带where的delete ：只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete语句为DML（data maintain Language),这个操作会被放到 rollback segment中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop是DLL（data define language),操作立即生效，原数据不放到 rollback segment中，不能回滚

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用delete且注意结合where来约束影响范围。回滚段要足够大。要删除表用drop;若想保留表而将表中数据删除，如果于事务无关，用truncate即可实现。如果和事务有关，或老师想触发trigger,还是用delete。

（10） Truncate table 表名 速度快,而且效率高,因为:  
truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

## 存储过程与触发器的区别

触发器与存储过程非常相似，触发器也是SQL语句集，**两者唯一的区别是触发器不能用EXECUTE语句调用，而是在用户执行Transact-SQL语句时自动触发（激活）执行。触发器是在一个修改了指定表中的数据时执行的存储过程。**通**常通过创建触发器来强制实现不同表中的逻辑相关数据的引用完整性和一致性。**由于用户不能绕过触发器，所以可以用它来强制实施复杂的业务规则，以确保数据的完整性。触发器不同于存储过程，**触发器主要是通过事件执行触发而被执行的**，而**存储过程可以通过存储过程名称名字而直接调用**。当对某一表进行诸如UPDATE、INSERT、DELETE这些操作时，SQLSERVER就会自动执行触发器所定义的SQL语句，从而确保对数据的处理必须符合这些SQL语句所定义的规则。

## 主键 超键 候选键 外键

**主 键：**

数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。**一个数据列只能有一个主键**，且主键的取值不能缺失，即不能为空值（Null）。

**超 键：**

在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。**超键包含候选键和主键。**

**候选键：**

是**最小超键**，即没有冗余元素的超键。

**外 键：**

在一个表中存在的**另一个表的主键**称此表的外键。

## 乱码问题

set names utf8/gbk;可以解决乱码问题。

character\_set\_client，character\_set\_connection，character\_set\_results均会变为相应的编码。

信息输入路径：client→connection→server；信息输出路径：server→connection→results。每个路径都需要经过三次改变字符集的编码，但是如果字符集编码不兼容，转化过程不可逆，就会出现乱码。如latin1和utf8不兼容。

## 存储引擎

存储引擎也叫表类型，是指的是数据表的存储机制，索引方案等配套相关功能。

默认的存储引擎通过my.ini文件配置Default-storage-engine=INNODB

在创建表或者编辑表时，可以指定存储引擎。

alter table tbl\_name engine myisam;

create table tbl\_name (

) engine myisam character set utf-8;

Innodb和myisam区别：

保存文件时，myisam，一个表，三个文件。

Tbl\_name.frm 结构

Tbl\_name.myd数据

Tbl\_name.myi索引

Innodb：一个表一个文件，tb\_name.frm结构。所有的innodb表都使用相同的innodb存储表空间保存数据和索引。（ibd文件）

数据和索引的保存的方式不同：myisam 是分开保存的，innodb是保存到表空间。

Myisam支持索引压缩，innodb索引和数据是绑定保存不压缩，体积大。

Innodb很多时候是行级锁，而myisam是表级锁，innodb并发高。

Innodb支持事务，外键，数据完整性约束要强。而myisam不支持。

Innodb中不保存表中的具体行数，也就是说select count(\*) from table时，innodb要扫描一遍表来计算有多少行，但是myisam只要简单的读出保存好的行数即可。

对于auto\_increment类型的字段，innodb中必须包含只有该字段的索引，但是在myisam表中，可以和其他字段一起建立联合索引。

## 高可用

http://gitbook.cn/books/583c1335c7f2666319396f7f/index.html

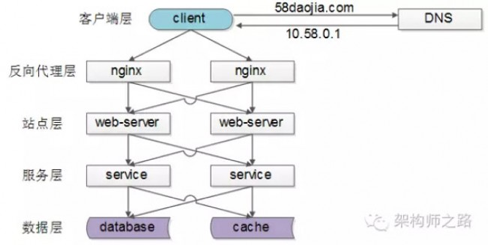
## 负载均衡

负载均衡（Load Balance）是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，**负载均衡的关键在于【均匀】**。常见互联网分布式架构如上，分为客户端层、反向代理nginx层、站点层、服务层、数据层。

**什么是负载均衡**

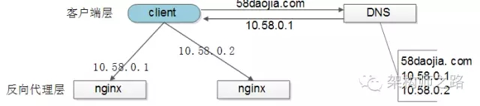
负载均衡(Load Balance)是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

**常见的负载均衡方案**



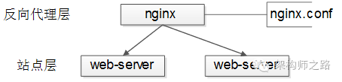
常见互联网分布式架构如上，分为客户端层、反向代理nginx层、站点层、服务层、数据层。可以看到，每一个下游都有多个上游调用，只需要做到，**每一个上游都均匀访问每一个下游，就能实现“将请求/数据【均匀】分摊到多个操作单元上执行”。**

**【客户端层->反向代理层】的负载均衡**



【客户端层】到【反向代理层】的负载均衡，是**通过“DNS轮询”实现**的：DNS-server对于一个域名配置了多个解析ip，每次DNS解析请求来访问DNS-server，会轮询返回这些ip，保证每个ip的解析概率是相同的。这些ip就是nginx的外网ip，以做到每台nginx的请求分配也是均衡的。

**【反向代理层->站点层】的负载均衡**



【反向代理层】到【站点层】的负载均衡，是**通过“nginx”实现**的。通过修改nginx.conf，可以实现多种负载均衡策略：

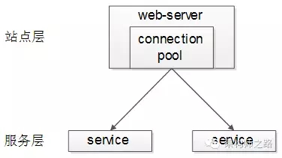
1)请求轮询：和DNS轮询类似，请求依次路由到各个web-server

2)最少连接路由：哪个web-server的连接少，路由到哪个web-server

3)ip哈希：按照访问用户的ip哈希值来路由web-server，只要用户的ip分布是均匀的，请求理论上也是均匀的，ip哈希均衡方法可以做到，同一个用户的请求固定落到同一台web-server上，此策略适合有状态服务，例如session(58沈剑备注：可以这么做，但强烈不建议这么做，站点层无状态是分布式架构设计的基本原则之一，session最好放到数据层存储)

4)…

**【站点层->服务层】的负载均衡**



【站点层】到【服务层】的负载均衡，是**通过“服务连接池”实现**的。

上游连接池会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。

上一篇文章《RPC-client实现细节》中有详细的负载均衡、故障转移、超时处理的细节描述，欢迎点击link查阅，此处不再展开。

**【数据层】的负载均衡**

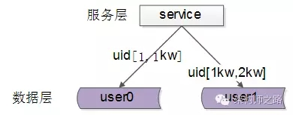
在数据量很大的情况下，由于数据层(db，cache)涉及数据的水平切分，所以数据层的负载均衡更为复杂一些，它分为“数据的均衡”，与“请求的均衡”。

数据的均衡是指：水平切分后的每个服务(db，cache)，数据量是差不多的。

请求的均衡是指：水平切分后的每个服务(db，cache)，请求量是差不多的。

业内常见的水平切分方式有这么几种：

**一、按照range水平切分**



每一个数据服务，存储一定范围的数据，上图为例：

user0服务，存储uid范围1-1kw

user1服务，存储uid范围1kw-2kw

这个方案的好处是：

(1)规则简单，service只需判断一下uid范围就能路由到对应的存储服务

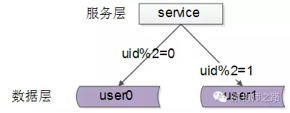
(2)数据均衡性较好

(3)比较容易扩展，可以随时加一个uid[2kw,3kw]的数据服务

不足是：

(1)请求的负载不一定均衡，一般来说，新注册的用户会比老用户更活跃，大range的服务请求压力会更大

**二、按照id哈希水平切分**



每一个数据服务，存储某个key值hash后的部分数据，上图为例：

user0服务，存储偶数uid数据

user1服务，存储奇数uid数据

这个方案的好处是：

(1)规则简单，service只需对uid进行hash能路由到对应的存储服务

(2)数据均衡性较好

(3)请求均匀性较好

不足是：

(1)不容易扩展，扩展一个数据服务，hash方法改变时候，可能需要进行数据迁移

**总结**

负载均衡(Load Balance)是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

(1)【客户端层】到【反向代理层】的负载均衡，是通过“DNS轮询”实现的

(2)【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的

(3)【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的

(4)【数据层】的负载均衡，要考虑“数据的均衡”与“请求的均衡”两个点，常见的方式有“按照范围水平切分”与“hash水平切分”

**Nginx/LVS/HAProxy负载均衡软件**

一般对负载均衡的使用是随着网站规模的提升根据不同的阶段来使用不同的技术。具体的应用需求还得具体分析，如果是中小型的Web应用，比如日PV小于1000万，用Nginx就完全可以了；如果机器不少，可以用DNS轮询，LVS所耗费的机器还是比较多的；大型网站或重要的服务，且服务器比较多时，可以考虑用LVS。

一种是通过硬件来进行进行，常见的硬件有比较昂贵的F5和Array等商用的负载均衡器，它的优点就是有专业的维护团队来对这些服务进行维护、缺点就是花销太大，所以对于规模较小的网络服务来说暂时还没有需要使用；另外一种就是类似于Nginx/LVS/HAProxy的基于[Linux](http://www.ha97.com/category/linux)的开源免费的负载均衡软件，这些都是通过软件级别来实现，所以费用非常低廉。

目前关于网站架构一般比较合理流行的架构方案：**Web前端采用Nginx/HAProxy+Keepalived作负载均衡器；后端采用**[MySQL](http://www.ha97.com/tag/mysql)**数据库一主多从和读写分离，采用LVS+Keepalived的架构。**当然要根据项目具体需求制定方案。  
下面说说各自的特点和适用场合。

一、Nginx

**Nginx的优点是：**

1、工作在网络的7层之上，可以针对http应用做一些分流的策略，比如针对域名、目录结构，它的**正则规则**比HAProxy更为强大和灵活，这也是它目前广泛流行的主要原因之一，Nginx单凭这点可利用的场合就远多于LVS了。  
2、**Nginx对网络稳定性的依赖非常小**，理论上能ping通就就能进行负载功能，这个也是它的优势之一；相反LVS对网络稳定性依赖比较大，这点本人深有体会；  
3、Nginx安装和配置比较简单，测试起来比较方便，它基本能把错误用日志打印出来。LVS的配置、测试就要花比较长的时间了，LVS对网络依赖比较大。  
3、可以承担高负载压力且稳定，在硬件不差的情况下一般能支撑几万次的并发量，负载度比LVS相对小些。  
4、Nginx可以通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点，不过其中缺点就是不支持url来检测。比如用户正在上传一个文件，而处理该上传的节点刚好在上传过程中出现故障，Nginx会把上传切到另一台服务器重新处理，而LVS就直接断掉了，如果是上传一个很大的文件或者很重要的文件的话，用户可能会因此而不满。  
5、Nginx不仅仅是一款优秀的负载均衡器/反向代理软件，它同时也是功能强大的Web应用服务器。LNMP也是近几年非常流行的web架构，在高流量的环境中稳定性也很好。  
6、Nginx现在作为Web反向加速缓存越来越成熟了，速度比传统的[Squid](http://www.ha97.com/tag/squid)服务器更快，可以考虑用其作为反向代理加速器。  
7、Nginx可作为中层反向代理使用，这一层面Nginx基本上无对手，唯一可以对比Nginx的就只有lighttpd了，不过lighttpd目前还没有做到Nginx完全的功能，配置也不那么清晰易读，社区资料也远远没Nginx活跃。  
8、Nginx也可作为静态网页和图片服务器，这方面的性能也无对手。还有Nginx社区非常活跃，第三方模块也很多。

淘宝的前端使用的Tengine就是基于nginx做的二次开发定制版。

Nginx常规的HTTP请求和响应流程图：

**Nginx的缺点是：**  
1、Nginx仅能支持http、[https](http://www.ha97.com/tag/https)和Email协议，这样就在适用范围上面小些，这个是它的缺点。  
2、**对后端服务器的健康检查，只支持通过端口来检测，不支持通过url来检测。**不支持Session的直接保持，但能通过ip\_hash来解决。

**二、LVS**

LVS：使用[Linux](http://www.ha97.com/tag/linux)内核集群实现一个高性能、高可用的负载均衡服务器，它具有很好的可伸缩性（Scalability)、可靠性（Reliability)和可管理性（Manageability)。

**LVS的优点是：**  
1、抗负载能力强、是工作在网络4层之上仅作分发之用，**没有流量的产生**，这个特点也决定了它在负载均衡软件里的性能最强的，对内存和cpu资源消耗比较低。  
2、配置性比较低，这是一个缺点也是一个优点，因为没有可太多配置的东西，所以并不需要太多接触，大大减少了人为出错的几率。  
3、工作稳定，因为其本身抗负载能力很强，自身有完整的双机热备方案，如LVS+Keepalived，不过我们在项目实施中用得最多的还是LVS/DR+Keepalived。  
4、无流量，LVS只分发请求，而流量并不从它本身出去，这点保证了均衡器IO的性能不会收到大流量的影响。  
5、应用范围比较广，因为LVS工作在4层，所以它几乎可以对所有应用做负载均衡，包括http、数据库、在线聊天室等等。

LVS DR(Direct Routing)模式的网络流程图：

**LVS的缺点是：**  
1、软件本身**不支持正则表达式处理，不能做动静分离**；而现在许多网站在这方面都有较强的需求，这个是Nginx/HAProxy+Keepalived的优势所在。  
2、如果是网站应用比较庞大的话，LVS/DR+Keepalived实施起来就比较复杂了，特别后面有Windows [Server](http://www.ha97.com/tag/server)的机器的话，如果实施及配置还有维护过程就比较复杂了，相对而言，Nginx/HAProxy+Keepalived就简单多了。

**三、HAProxy**

**HAProxy的特点是：**  
1、HAProxy也是支持虚拟主机的。  
2、HAProxy的优点能够补充Nginx的一些缺点，比如**支持Session的保持，Cookie的引导；同时支持通过获取指定的url来检测后端服务器的状态。**  
3、HAProxy跟LVS类似，本身就只是一款负载均衡软件；单纯从效率上来讲HAProxy会比Nginx有更出色的负载均衡速度，在并发处理上也是优于Nginx的。  
4、HAProxy**支持TCP协议的负载均衡转发**，可以对MySQL读进行负载均衡，对后端的MySQL节点进行检测和负载均衡，大家可以用LVS+Keepalived对MySQL主从做负载均衡。  
5、HAProxy负载均衡策略非常多，HAProxy的负载均衡算法现在具体有如下8种：  
① roundrobin，表示简单的轮询，这个不多说，这个是负载均衡基本都具备的；  
② static-rr，表示根据权重，建议关注；  
③ leastconn，表示最少连接者先处理，建议关注；  
④ source，表示根据请求源IP，这个跟Nginx的IP\_hash机制类似，我们用其作为解决session问题的一种方法，建议关注；  
⑤ ri，表示根据请求的URI；  
⑥ rl\_param，表示根据请求的URl参数’balance url\_param’ requires an URL parameter name；  
⑦ hdr(name)，表示根据HTTP请求头来锁定每一次HTTP请求；  
⑧ rdp-cookie(name)，表示根据据cookie(name)来锁定并哈希每一次TCP请求。

**四、总结**

**Nginx和LVS对比的总结：**  
1、Nginx工作在网络的7层，所以它可以针对http应用本身来做分流策略，比如针对域名、目录结构等，相比之下LVS并不具备这样的功能，所以Nginx单凭这点可利用的场合就远多于LVS了；但Nginx有用的这些功能使其可调整度要高于LVS，所以经常要去触碰触碰，触碰多了，人为出问题的几率也就会大。  
2、Nginx对网络稳定性的依赖较小，理论上只要ping得通，网页访问正常，Nginx就能连得通，这是Nginx的一大优势！Nginx同时还能区分内外网，如果是同时拥有内外网的节点，就相当于单机拥有了备份线路；LVS就比较依赖于网络环境，目前来看服务器在同一网段内并且LVS使用direct方式分流，效果较能得到保证。另外注意，LVS需要向托管商至少申请多一个ip来做Visual IP，貌似是不能用本身的IP来做VIP的。要做好LVS管理员，确实得跟进学习很多有关网络通信方面的知识，就不再是一个HTTP那么简单了。  
3、Nginx安装和配置比较简单，测试起来也很方便，因为它基本能把错误用日志打印出来。LVS的安装和配置、测试就要花比较长的时间了；LVS对网络依赖比较大，很多时候不能配置成功都是因为网络问题而不是配置问题，出了问题要解决也相应的会麻烦得多。  
4、Nginx也同样能承受很高负载且稳定，但负载度和稳定度差LVS还有几个等级：Nginx处理所有流量所以受限于机器IO和配置；本身的bug也还是难以避免的。  
5、Nginx可以检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点。目前LVS中 ldirectd也能支持针对服务器内部的情况来监控，但LVS的原理使其不能重发请求。比如用户正在上传一个文件，而处理该上传的节点刚好在上传过程中出现故障，Nginx会把上传切到另一台服务器重新处理，而LVS就直接断掉了，如果是上传一个很大的文件或者很重要的文件的话，用户可能会因此而恼火。  
6、Nginx对请求的异步处理可以帮助节点服务器减轻负载，假如使用apache直接对外服务，那么出现很多的窄带链接时apache服务器将会占用大 量内存而不能释放，使用多一个Nginx做apache代理的话，这些窄带链接会被Nginx挡住，apache上就不会堆积过多的请求，这样就减少了相当多的资源占用。这点使用squid也有相同的作用，即使squid本身配置为不缓存，对apache还是有很大帮助的。  
7、Nginx能支持http、https和email（email的功能比较少用），LVS所支持的应用在这点上会比Nginx更多。在使用上，一般最前端所采取的策略应是LVS，也就是DNS的指向应为LVS均衡器，LVS的优点令它非常适合做这个任务。重要的ip地址，最好交由LVS托管，比如数据库的 ip、webservice服务器的ip等等，这些ip地址随着时间推移，使用面会越来越大，如果更换ip则故障会接踵而至。所以将这些重要ip交给 LVS托管是最为稳妥的，这样做的唯一缺点是需要的VIP数量会比较多。Nginx可作为LVS节点机器使用，一是可以利用Nginx的功能，二是可以利用Nginx的性能。当然这一层面也可以直接使用squid，squid的功能方面就比Nginx弱不少了，性能上也有所逊色于Nginx。Nginx也可作为中层代理使用，这一层面Nginx基本上无对手，唯一可以撼动Nginx的就只有lighttpd了，不过lighttpd目前还没有能做到 Nginx完全的功能，配置也不那么清晰易读。另外，中层代理的IP也是重要的，所以中层代理也拥有一个VIP和LVS是最完美的方案了。具体的应用还得具体分析，如果是比较小的网站（日PV小于1000万），用Nginx就完全可以了，如果机器也不少，可以用DNS轮询，LVS所耗费的机器还是比较多的；大型网站或者重要的服务，机器不发愁的时候，要多多考虑利用LVS。

**现在对网络负载均衡的使用是随着网站规模的提升根据不同的阶段来使用不同的技术：**

第一阶段：利用Nginx或HAProxy进行单点的负载均衡，这一阶段服务器规模刚脱离开单服务器、单数据库的模式，需要一定的负载均衡，但是仍然规模较小没有专业的维护团队来进行维护，也没有需要进行大规模的网站部署。这样利用Nginx或HAproxy就是第一选择，此时这些东西上手快， 配置容易，在七层之上利用HTTP协议就可以。这时是第一选择。

第二阶段：随着网络服务进一步扩大，这时单点的Nginx已经不能满足，这时使用LVS或者商用Array就是首要选择，Nginx此时就作为LVS或者Array的节点来使用，具体LVS或Array的是选择是根据公司规模和预算来选择，**Array的应用交付功能非常强大，本人在某项目中使用过，性价比也远高于F5，商用首选！**但是一般来说这阶段相关人才跟不上业务的提升，所以购买商业负载均衡已经成为了必经之路。

第三阶段：这时网络服务已经成为主流产品，此时随着公司知名度也进一步扩展，相关人才的能力以及数量也随之提升，这时无论从开发适合自身产品的定制，以及降低成本来讲开源的LVS，已经成为首选，这时LVS会成为主流。  
最终形成比较理想的基本架构为：**Array/LVS — Nginx/Haproxy — Squid/Varnish — AppServer**

## 数据库分库分表

<http://blog.csdn.net/lihuayong/article/details/42044235>

<http://www.cnblogs.com/miketwais/articles/mysql_partition.html>

<http://www.2cto.com/database/201503/380348.html>

<http://mp.weixin.qq.com/s/ezD0CWHAr0RteC9yrwqyZA>

<http://blog.csdn.net/heirenheiren/article/details/7896546>

### 数据表水平切分的ID规则

数据表水平切分后，也会带来一些问题，如切分规则的设计 切分后的访问路由 切分后的主键的全局唯一性等。

* 通过应用程序生成一个GUID，然后和数据一起插入切分后的集群。优点是维护简单，实现也容易。缺点是应用的计算成本较大，且GUID比较常，占用数据库存储空间较大，涉及到应用的开发。   
  --------------------------------------------------------------------------------  
  说明：主要优势是简单，缺点是浪费存储空间，GUID：32字节，100W记录，多32兆，如果是int，4字节，会少28M。如果有外键关联，会浪费更多。
* 通过独立的应用程序事先在数据库中生成一系列唯一的 ID，各应用程序通过接口或者自己去读取再和数据一起插入到切分后的集群中。优点是全局唯一主键简单，维护相对容易。缺点是实现复杂，需要应用开发。   
  --------------------------------------------------------------------------------  
  说明：ID表要频繁查和频繁更新，插入数据时，影响性能。
* 通过中心数据库服务器利用数据库自身的自增类型（如 MySQL的 auto\_increment 字段），或者自增对象（如 Oracle 的 Sequence）等先生成一个唯一 ID 再和数据一起插入切分后的集群。优点是？好像没有特别明显的优点。缺点是实现较为复杂，且整体可用性维系在这个中心数据库服务器上，一旦这里crash 了，所有的集群都无法进行插入操作，涉及到应用开发。  
  --------------------------------------------------------------------------------  
  说明：不推荐。
* 通过集群编号加集群内的自增（auto\_increment类型）两个字段共同组成唯一主键。优点是实现简单，维护也比较简单，对应用透明。缺点是引用关联操作相对比较复杂，需要两个字段，主键占用空间较大，在使用 InnoDB 的时候这一点的副作用很明显。   
  ---------------------------------------------------------------------------------  
  说明：虽然是两个字段，但是这方式存储空间最小，仅仅多了一个smallint两个字节，100W也就多2M。
* 通过设置每个集群中自增 ID 起始点（auto\_increment\_offset），将各个集群的ID进行绝对的分段来实现全局唯一。当遇到某个集群数据增长过快后，通过命令调整下 一个 ID 起始位置跳过可能存在的冲突。优点是实现简单，且比较容易根据 ID 大小直接判断出数据处在哪个集群，对应用透明。缺点是维护相对较复杂，需要高度关注各个集群 ID 增长状况。   
  ---------------------------------------------------------------------------------  
  说明：段满了，调整太麻烦。
* 通过设置每个集群中自增 ID 起始点（auto\_increment\_offset）以及 ID 自增步长（auto\_increment\_increment），让目前每个集群的起始点错开 1，步长选择大于将来基本不可能达到的切分集群数，达到将 ID 相对分段的效果来满足全局唯一的效果。优点是实现简单，后期维护简单，对应用透明。缺点是第一次设置相对较为复杂。   
  ---------------------------------------------------------------------------------  
  说明：会重合吧？

上面的各种方案没有考虑，外键关联时，怎么获取数据的问题，比如博客帖子里面保存了用户ID是1，那么用户ID为1的用户存在哪个数据库中呢？通过ID是无法知道的。

1. 因此，水平切分后，要么，就像上文的第4中，就是集群编号+用户ID做为主外键.则其他表外键关联时就必须关联两个字段：集群编号、对象编号。集群ID可以用TinyInt等只有1个字节的，这样100W只多1M。
2. 另外的一个方案是改进版本的第5个方案。分段是可行的，int不是4字节，32位么。那么拿出几位来做集群编号。2^6为集群编号（前6位，共64段），2^26为对象ID区域，就是说  
       假如集群编号是n（0-n），2^26是常数G，那么开始段为n\*G，结束段为(n+1)\*G-1  
   通过这样处理之后，那么外键关联时，只要取ID的前6位，就知道集群编号了。根据集群编号才能访问数据库啊。每段有2^26，有6kw的数据量，因为64个集群，所以，基本上是够用了，如果每个库的数据量太大，ID数据类型改为BigInt（8字节）之类的也就可以了，要尽量避免数据不够的问题，当然单表超过这个数据量，估计还是增加集群机器数量比较妥当。

### 数据表水平切分的方法

1 根据id进行切分

Db1对应id为1-1000 db2对应id为1001-2000.。。。

优点：可部分迁移

缺点：数据分布不均匀

2 hash取模分区

对id进行hash，然后对数据库的数量进行取模

优点：数据分布均匀

缺点：数据迁移比较麻烦，不能按照机器的性能分摊数据

3 在认证库中保存数据库的配置

单独建立一个数据库，保存id到db的映射关系，每次访问数据库的时候都先查询这个数据库，以得到具体的db信息，然后再进行我们需要的查询操作。

优点：灵活性强，一对一

缺点：多了一次查询，性能下降

### 基本思想

Sharding（分片分表）的基本思想就要把一个[数据库](http://lib.csdn.net/base/mysql)切分成多个部分放到不同的数据库(server)上，从而缓解单一数据库的性能问题。对于海量数据的数据库，如果是因为表多而数据多，这时候适合使用垂直切分，即把关系紧密（比如同一模块）的表切分出来放在一个服务器上。如果表并不多，但每张表的数据非常多，这时候适合水平切分，即把表的数据按某种规则（比如按ID散列）切分到多个数据库(server)上。根据实际情况做出选择，也可能会综合使用垂直与水平切分。

## Innodb myisam merge存储引擎

<http://blog.csdn.net/qmhball/article/details/7499432?reload>