悲观锁和乐观锁：

悲观锁:

与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，

所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，

所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。

说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。

例如：行锁 表锁 读锁 写锁都是操作之前先上锁

乐观锁：

https://www.cnblogs.com/qlqwjy/p/7798266.html

使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。

何谓数据版本？即为数据增加一个版本标识，一般是通过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值加一

。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，

如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据

Mycat:

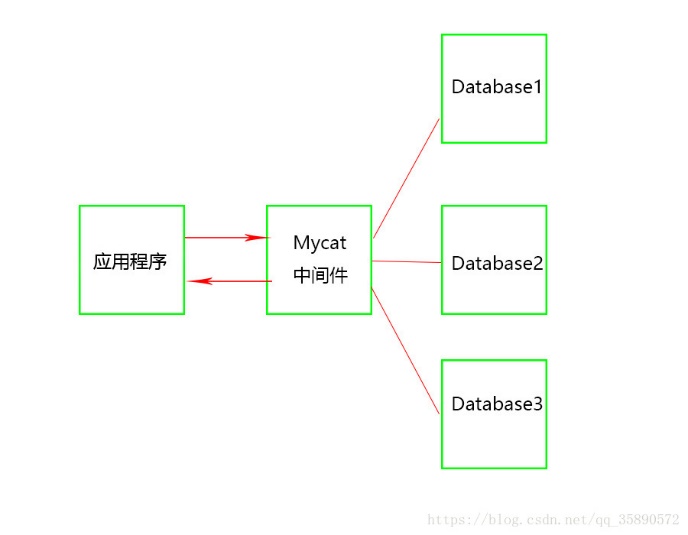
介绍：

MyCat是一个开源的分布式数据库系统，是一个实现了MySQL协议的服务器，前端用户可以把它看作是一个数据库代理，用MySQL客户端工具和命令行访问，而其后端可以用MySQL原生协议与多个MySQL服务器通信，也可以用JDBC协议与大多数主流数据库服务器通信，其核心功能是分表分库，即将一个大表水平分割为N个小表，存储在后端MySQL服务器里或者其他数据库里

原理：

Mycat的原理中最重要的一个动词是“拦截”，它拦截了用户发送过来的SQL语句，首先对SQL语句做了一些特定的分析：如分片分析、路由分析、读写分离分析、缓存分析等，然后将此SQL发往后端的真实数据库，并将返回的结果做适当的处理，最终再返回给用户。

图片里，应用程序不再直接访问数据库，而是访问Mycat，由Mycat与数据库交互，数据库数据返回给Mycat，Mycat再返回给应用程序。三个Database才是真正的数据库，又称为三个节点，也称为三个分片。



mycat三大配置文件？

schema.xml:定义逻辑库，逻辑表，分片规则

<schema> 逻辑库

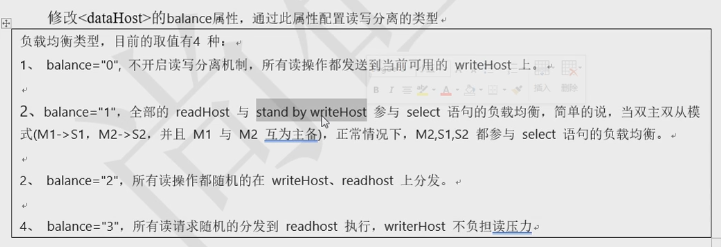
<dataNode> 具体库

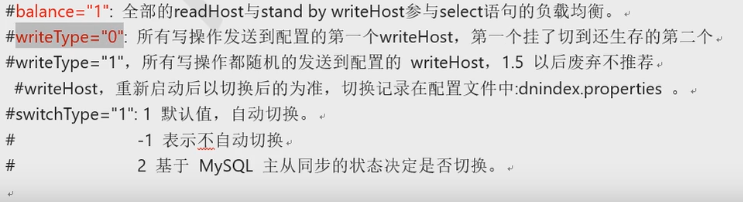
<dataHost> 读写数据库

<writebeat>写库

<readbeat> 读库

读写分离？<dataHost> balance 属性



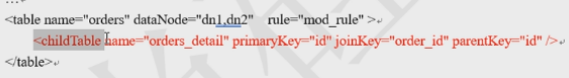


分库：<table name="customer" dataNode="dn2" />

分表：<table name="order" dataNode="dn1,dn2" rule=”PartitionByMod

” />

ER表：相当于join查询 父表和子表分片规则相同 primaryKey joinKey parentKey



全局表：具体数据库都有这数据



全局序列：

方式1：mycat 以时间戳作为id

缺点：时间戳作为id太长

方式2：序列表保存在redis中（利用incr），内存操作快

rule.xml：具体分片策略

取模分片：PartitionByMod

枚举分片：PartitionByFileType

范围分片：PartitionByLong

日期分片：PartitionByDate

server.xml：用户白名单，黑名单 对库，表的操作权限

sql优化：

1.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。

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

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**

3.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。  
  
4.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'

5.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)

6.下面的查询也将导致全表扫描：

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

若要提高效率，可以考虑全文检索。  
  
7.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

可以改为强制查询使用索引：

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

.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**

9.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'

10.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。  
  
11.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。  
  
12.不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where **1**=**0**

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：  
create table #t(…)

13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。  
  
14.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。  
  
15.select count(\*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

16.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。  
  
17.应尽可能的避免更新 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。  
  
18.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。  
  
19.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。  
  
20.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。  
  
21.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）。  
  
22. 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。  
  
23.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。  
  
24.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。  
  
25.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。  
  
26.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。  
  
27.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。  
  
28.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。  
  
29.尽量避免大事务操作，提高系统并发能力。  
  
30.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

**实际案例分析**：拆分大的 DELETE 或INSERT 语句，批量提交SQL语句  
　　如果你需要在一个在线的网站上去执行一个大的 DELETE 或 INSERT 查询，你需要非常小心，要避免你的操作让你的整个网站停止相应。因为这两个操作是会锁表的，表一锁住了，别的操作都进不来了。  
　　Apache 会有很多的子进程或线程。所以，其工作起来相当有效率，而我们的服务器也不希望有太多的子进程，线程和数据库链接，这是极大的占服务器资源的事情，尤其是内存。  
　　如果你把你的表锁上一段时间，比如30秒钟，那么对于一个有很高访问量的站点来说，这30秒所积累的访问进程/线程，数据库链接，打开的文件数，可能不仅仅会让你的WEB服务崩溃，还可能会让你的整台服务器马上挂了。  
　　所以，如果你有一个大的处理，你一定把其拆分，使用 LIMIT oracle(rownum),sqlserver(top)条件是一个好的方法。下面是一个mysql示例：

[复制代码](javascript:void(0);)

while(1){

　　//每次只做1000条

　　 mysql\_query(“delete from logs where log\_date <= ’2012-11-01’ limit 1000”);

　　if(mysql\_affected\_rows() == 0){

　　 　　//删除完成，退出！

　　 　　break；

　　}

//每次暂停一段时间，释放表让其他进程/线程访问。

usleep(50000)

}

[复制代码](javascript:void(0);)

索引优化详解：<https://blog.csdn.net/HXNLYW/article/details/82979088>