MySQL

- 基础知识：
  - 数据库范式：
    - 1NF - 原子性
      1NF是对属性的原子性，要求属性具有原子性，不可再分解；
      表：字段1、 字段2(字段2.1、字段2.2)、字段3 .....
      如学生（学号，姓名，性别，出生年月日），如果认为最后一列还可以再分成（出生年，出生月，出生日），它就不是一范式了，否则就是；
      - 第一范式（1NF）无重复的列
    - 2NF - 惟一性
      2NF是对记录的惟一性，要求记录有惟一标识，即实体的惟一性，即不存在部分依赖；
      表：学号、课程号、姓名、学分；
      这个表明显说明了两个事务:学生信息, 课程信息;由于非主键字段必须依赖主键，这里学分依赖课程号，姓名依赖与学号，所以不符合二范式。
      可能会存在问题:
      数据冗余:，每条记录都含有相同信息;
      删除异常：删除所有学生成绩，就把课程信息全删除了;
      插入异常：学生未选课，无法记录进数据库;
      更新异常：调整课程学分，所有行都调整。
      正确做法:
      学生：Student(学号, 姓名);
      课程：Course(课程号, 学分);
      选课关系：StudentCourse(学号, 课程号, 成绩)。
      - 属性完全依赖于主键 [消除部分子函数依赖]
    - 3NF - 冗余性
      3NF是对字段的冗余性，要求任何字段不能由其他字段派生出来，它要求字段没有冗余，即不存在传递依赖；
      表: 学号, 姓名, 年龄, 学院名称, 学院电话
      因为存在依赖传递: (学号) → (学生)→(所在学院) → (学院电话) 。
      可能会存在问题：
      数据冗余:有重复值;
      更新异常：有重复的冗余信息，修改时需要同时修改多条记录，否则会出现数据不一致的情况 。
      正确做法:
      学生：(学号, 姓名, 年龄, 所在学院);
      学院：(学院, 电话)。
      - 属性不依赖于其它非主属性 [消除传递依赖]
    - BCNF
      - 鲍依斯-科得范式（BCNF是3NF的改进形式）
    - 反范式化
      没有冗余的数据库设计可以做到。但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。具体做法是：在概念数据模型设计时遵守第三范式，降低范式标准的工作放到物理数据模型设计时考虑。降低范式就是增加字段，允许冗余，达到以空间换时间的目的。
      『例』：有一张存放商品的基本表，“金额”这个字段的存在，表明该表的设计不满足第三范式，因为“金额”可以由“单价”乘以“数量”得到，说明“金额”是冗余字段。但是，增加“金额”这个冗余字段，可以提高查询统计的速度，这就是以空间换时间的作法。
      在Rose 2002中，规定列有两种类型：数据列和计算列。“金额”这样的列被称为“计算列”，而“单价”和“数量”这样的列被称为“数据列”。
  - 数据库锁分类：
    - 按封锁类型分类：
      - 乐观锁
        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1。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。
        举例:
        1、数据库表三个字段，分别是id、value、version
        select id,value,version from TABLE where id = #{id}
        2、每次更新表中的value字段时，为了防止发生冲突，需要这样操作
        update TABLE
        set value=2,version=version+1
        where id=#{id} and version=#{version}
      - 悲观锁
        - 1）排他锁：（又称写锁，X锁）
        - 2）共享锁：（又称读取，S锁）
    - 按封锁的数据粒度分类如下：
      - 1）表级锁定（table-level）：
        - 开销小，加锁快；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。
      - 2）页级锁定（page-level）：（MySQL特有）
        - 开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。
      - 3）行级锁定（row-level）：

        - 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，并发度也最高。
      - 间隙锁
    - 活锁与死锁
- 数据库事务：

  - ACID
    - 原子性（ Atomicity ）、一致性（ Consistency ）、隔离性（ Isolation ）和持续性（ Durability ）

      1 、原子性。事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做
      2 、一致性。事 务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。
      3 、隔离性。一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。
      4 、持续性。也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。
    - **事务引发的调度问题：**
      - 1.脏读（dirty read）

        A事务读取B事务尚未提交的更改数据，并在这个数据基础上操作。如果B事务回滚，**那么A事务读到的数据根本不是合法的，称为脏读。在oracle中，由于有version控制，不会出现脏读。**
        例子：事务三中售出一张票，修改了数据库31变30，可是事务还没提交，事务一读了修改了的数据（30），但是事务三被中断撤销了，比如存钱修改数据库后，在返回数据给客户时出现异常，那么事务三就是不成功的，数据会变回31。可是事务一读了一个不正确的数据。
      - 2.不可重复读（unrepeatable read）

        A事务读取了B事务已经提交的更改数据。比如A事务第一次读取数据，然后B事务更改该数据并提交，A事务再次读取数据，两次读取的数据不一样。
        例子：事务一读取后，事务三对31张票修改，可是事务一中，有再次读这张票的SQL语句，那么事务一得到的跟第一次不同的值（31张票就可能变成30张了）。
      - 3.幻读（phantom read）

        A事务读取了B事务已经提交的新增数据。注意和不可重复读的区别，这里是新增与删除，不可重复读是更改。这两种情况对策是不一样的，对于不可重复读，只需要采取行级锁防止该记录数据被更改或删除，然而对于幻读必须加表级锁，防止在这个表中新增一条数据。
        例子：事务一按一定条件读取31条记录，但是后面还有一些步骤需要再次读取此数据校验，所以没提交事务，可这个时候事务三（31条记录变30条记录）执行成功，卖出了票。那当事务一再次读的时候，就读到了30。这样读到的数据就不是最新的了。
      - 4.丢失更新
        - A事务撤销时，把已提交的B事务的数据覆盖掉。
      - 5.覆盖更新
        - A事务提交时，把已提交的B事务的数据覆盖掉。
    - 三级封锁协议：
    - 事务隔离级别：

      SQL标准定义了4类隔离级别，包括了一些具体规则，用来限定事务内外的哪些改变是可见的，哪些是不可见的。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。
      - 1.读未提交(Read Uncommited，RU)

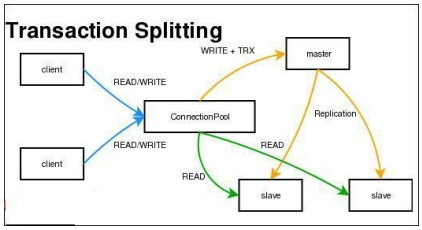
        在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。
      - 2.读提交(Read commited，RC)

        这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。
      - 3.可重复读(Repeatable Read, RR)

        这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。
        InnoDB和Falcon存储引擎通过多版本并发控制（**MVCC，Multiversion Concurrency Control）机制解决了该问题。**
        - MVCC
      - 4.串行化(Serializable)

        这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。
        这四种隔离级别采取不同的锁类型来实现，若读取的是同一个数据的话，就容易发生问题。例如：
        脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。
        不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。
        幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。
- **主从热备 & 读写分离**
  - 读写分离基础知识：
    - 1. 什么是读写分离：

      MySQL Proxy最强大的一项功能是实现“读写分离(Read/Write Splitting)”。基本的原理是让主数据库处理事务性查询，而从数据库处理SELECT查询。数据库复制被用来把事务性查询导致的变更同步到集群中的从数据库。 当然，主服务器也可以提供查询服务。使用读写分离最大的作用无非是环境服务器压力。
      -



- **2. 读写分离的好处：**
  - 1.增加冗余
  - 2.增加了机器的处理能力
  - 3.对于读操作为主的应用，使用读写分离是最好的场景，因为可以确保写的服务器压力更小，而读又可以接受点时间上的延迟。
- 3. 为什么读写分离能提升性能：
  - 1.物理服务器增加，负荷增加
  - 2.主从只负责各自的写和读，极大程度的缓解X锁和S锁争用
  - 3.从库可配置myisam引擎，提升查询性能以及节约系统开销
  - 4.从库同步主库的数据和主库直接写还是有区别的，通过主库发送来的binlog恢复数据，但是，最重要区别在于主库向从库发送binlog是异步的，从库恢复数据也是异步的
  - 5.读写分离适用于读远大于写的场景，如果只有一台服务器，当select很多时，update和delete会被这些select访问中的数据堵塞，等待select结束，并发性能不高。

    对于写和读比例相近的应用，应该部署双主相 ...
  - 6.可以在从库启动是增加一些参数来提高其读的性能

    当然这些设置也是需要根据具体业务需求来定得，不一定能用上
    - --skip-innodb
    - --skip-bdb
    - --low-priority-updates
    - --delay-key-write=ALL
  - 7.分摊读取。

    假如我们有1主3从，不考虑上述1中提到的从库单方面设置，假设现在1分钟内有10条写入，150条读取。那么，1主3从相当于共计40条写入，而读取总数没变，因此平均下来每台服务器承担了10条写入和50条读取（主库不承担读取操作）。因此，虽然写入没变，但是读取大大分摊了，提高了系统性能。另外，当读取被分摊后，又间接提高了写入的性能。所以，总体性能提高了，说白了就是拿机器和带宽换性能。MySQL官方文档中有相关演算公式：官方文档 见6.9FAQ之“MySQL复制能够何时和多大程度提高系统性能”
  - 8.MySQL复制另外一大功能是增加冗余，提高可用性，当一台数据库服务器宕机后能通过调整另外一台从库来以最快的速度恢复服务，因此不能光看性能，也就是说1主1从也是可以的。
- MySql 主从热备份工作原理
  - 简单的说：就是主服务器上执行过的操作会被保存在binLog日志里面，从服务器把他同步过来，然后重复执行一遍，那么它们就能一直同步啦
  - 整体上来说，复制有3个步骤：
    - 作为主服务器的Master，会把自己的每一次改动（每条sql语句）都记录到二进制日志Binarylog中。
    - 作为从服务器Slave， 会用master上的账号登陆到 master上，读取master的Binarylog,写入到自己的中继日志Relaylog。
    - 然后从服务器自己的sql线程会负责读取这个中继日志，并执行一遍
    - END

      到这里主服务器上的更改就同步到从服务器上了，这种复制和重复都是mysql自动实现的，我们只需要配置即可。
      整个过程，MySQL使用**3个线程来执行复制同步功能，其中两个线程(Sql线程和IO线程)在从服务器，另外一个线程(IO线程)在主服务器。**
      主-主互相复制实际只是把上面的例子反过来再做一遍。
      所以我们以这个例子介绍原理。
  - 主从服务器的配置：
    - 主服务器
      - 配置要记录在binLog里面的数据库，同时设置一个给从服务器登录的账号
    - 从服务器
      - 配置relayLog，配置主服务器的地址和端口，还有主服务器给的登录账号
- 主从备份的具体实现：
  - 在主服务器上设置给从服务器登录用的账号

    rant replication slave on *.* to 'relay_user1'@'192.168.1.168' identified by 'pass123456';
    注意：
    从服务器地址为：192.168.1.168，换成你的从机器的IP地址，这样就只允许从服务器登录,相对安全些。
    用户名为： relay_user1
    密码为： pass123456，自己按需要修改。
  - 修改MySQL配置文件

如果有Workbench，可以在左边的Navigator栏找到Options File,然后进行配置，如果想直接修改配置文件，文件的位置请看结尾的后记3.
主服务器的my.cnf配置
log-bin=master-a-bin
binlog-format=ROW //二进制日志的格式，有row、statement和mixed三种类型
server-id=1//要求各个服务器的这个id必须不一样
binlog-do-db=test //我们想让主服务器记录下操作的数据库。好让从服务器去复制的。
auto_increment_offset = 1 //设置AUTO_INCREMENT起点，关于这个看后记4
auto_increment_increment = 10 //设置AUTO_INCREMENT增量
//下面是一些别的配置，你可以跳过不看的
//
gtid-mode=on //启用GTID,可看结尾的后记2说明
enforce-gtid-consistency=true //启用GTID
master-info-repository=TABLE//默认是file，选择table方式保存
relay-log-info-repository=TABLE//默认是file，选择table方式保存
sync-master-info=1 //实时同步
slave-parallel-workers=2 //设定从服务器的SQL线程数；0表示关闭多线程复制功能
binlog-checksum=CRC32 //日志校验
master-verify-checksum=1//启用校验
slave-sql-verify-checksum=1//启用校验
innodb_flush_log_at_trx_commit=1 //每N次事务提交或事务外的指令都需要把日志写入（flush）硬盘
sync_binlog=1 //This makes MySQL synchronize the binary log’s contents to disk each time it commits a transaction
从服务器的my.cnf配置
log-bin=master-a-bin.log
binlog-format=mixed//请保持这两个一致
server-id=2
auto_increment_offset = 1
auto_increment_increment = 10
log-slave-updates=true //log-slave-updates 意思是，中继日志执行之后，这些变化是否需要计入自己的binarylog。 当你的从服务器需要作为另外一个服务器的主服务器的时候需要打开。 就是双主互相备份，或者多主循环备份。 我们这里需要， 所以打开。
//配置主服务器相关的信息
replicate-do-db = test
report-host = 192.168.0.101
report-user = repl_user
report-password = 112122
report-port = 3306
//别的一些配置,可以跳过不看
sync_binlog=1
gtid-mode=on
enforce-gtid-consistency=true
master-info-repository=TABLE
relay-log-info-repository=TABLE
sync-master-info=1
slave-parallel-workers=2
binlog-checksum=CRC32
master-verify-checksum=1
slave-sql-verify-checksum=1
binlog-rows-query-log-events=1
这样就算配置完咯，我不会告诉你我是用workbench改的，然后在apply的时候，把它复制下来粘贴到这里的，需要注意的是，如果你数据库原本就有表创建了，从服务器没有这个表的话，
请手动创建一个，
请手动创建一个，
请手动创建一个。
因为这个建表语句在这个同步操作前就执行过，没保存啊。我已经帮你试过了，下面试错误日志里面的信息.
2015-10-05 22:00:32 2628 [Warning] Slave: Table 'test.user' doesn't exist Error_code: 1146
2015-10-05 22:00:32 2628 [ERROR] Error running query, slave SQL thread aborted. Fix the problem, and restart the slave SQL thread with "SLAVE START". We stopped at log 'bin0001.000003' position 151
另外需要说的是，如果遇到这样的错误，请改正后重启下。

- 重启同步

```
1
2
3
4
stop slave;
change master to master_host='192.168.0.101',master_user='repl_user',master_password='pass123456',master_auto_position=1;
start slave;
```

既然都配置好了，那就重启下，让配置生效
可以在主服务器打印下信息，看下有没启动成功先的
mysql> show master status ;
从服务器;
mysql> show slave status ;
如果打印的内容中，下面两句后面都为yes，那就恭喜你，正常运行咯
slave_io_running : yes
slave_sql_running : yes
Slave_IO_Running: 是否要从Master复制二进制数据
Slave_SQL_Running: 是否执行从Master复制过来的二进制数据
Slave_IO_Running和Slave_SQL_Running的值均为Yes时为同步开启;
前面已经说过了，从服务器会有两个线程的，一个IO线程和一个执行sql的线程。
说到这里，我不得不吐槽下在window下的cmd那个界面有多那么让人想死。
真的可以让人抓狂的。
然后试下在主服务器插入点数据，在从服务器能否同步回来？ 如果不能请打开errLog.err或者xxx.err之类的文件。

-

- W7用户：
  - 错误日志在：C:\ProgramData\MySQL\MySQL Server 5.6\data\你的电脑名.err

- **搭配问题：**
  - **1. 单一 master 和多个 slave**

    读多写少

    - 特点：
      - 多Slave之间并不相互通信，只能与master进行通信
      - 如果写操作较少，而读操作很多时，可以采取这种结构
      - 你可以将读操作分布到其它的slave，从而减小master的压力
    - **存在的问题：**
      - 当slave增加到一定数量时，slave对master的负载以及网络带宽都会成为一个严重的问题。
      - 这种结构虽然简单，但是，它却非常灵活，足够满足大多数应用需求。
    - 优化建议：
      - (1) 不同的slave扮演不同的作用(例如使用不同的索引，或者不同的存储引擎)；
      - (2) 用一个slave作为备用master，只进行复制；
      - (3) 用一个远程的slave，用于灾难恢复；
  - **2. 主动模式的 Master-Master (双主热备份)**

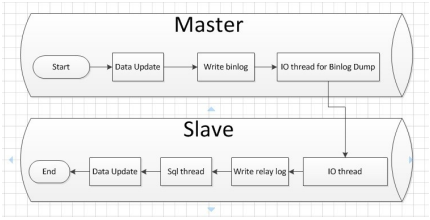
    关于如何做双主热备份，其实你可以理解为，原来的从服务器变成了主服务器，主服务器变成了从服务器，这样，他们就互为主从，变成双主热备份啦！！是不是？所以你只需要把配置文件copy一份，就好了。

    - 特点：
      - Master-Master复制的两台服务器，既是master，又是另一台服务器的slave
    - 用途：

      主动的Master-Master复制有一些特殊的用处
      - 地理上分布的两个部分都需要自己的可写的数据副本
    - 存在的问题 - 更新冲突：

      假设一个表只有一行(一列)的数据，其值为1，如果两个服务器分别同时执行如下语句：
      在第一个服务器上执行：
      mysql> UPDATE tbl SET col=col + 1;
      在第二个服务器上执行：
      mysql> UPDATE tbl SET col=col 2;
      那么结果是多少呢？一台服务器是4，另一个服务器是3，但是，这并不会产生错误。
      实际上，*MySQL并不支持其它一些DBMS支持的多主服务器复制(Multimaster Replication)，这是MySQL的复制功能很大的一个限制(多主服务器的难点在于解决更新冲突)，但是，如果你实在有这种需求，你可以采用MySQL Cluster，以及将Cluster和Replication结合起来，可以建立强大的高性能的数据库平台。
      但是，可以通过其它一些方式来模拟这种多主服务器的复制。
    - Mysql + keepalived 实现双主热备读写分离

      
      - 系统配置信息：
        - 系统：CentOS6.4_X86_64
        - 软件版本：Mysql-5.6.12,Keepalived-1.2.7
        - 环境简介：
        - 1.Master-A 192.168.1.168 (Mysql+Keepalived)
        - 2.Master-B 192.168.1.169 (Mysql+Keepalived)
        - 3.写入VIP 192.168.1.100 (168主，169从)
        - 4.读取VIP 192.168.1.200 (169主，168从)
      - 工作流程图
        -

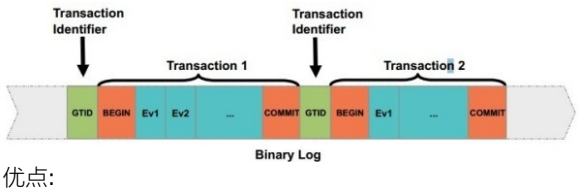


- 复制实现原理（适用于Mysql 5.5及之前的版本）：

  MySQL支持单向,异步复制，复制过程中一个服务器充当主服务器，而另外一个或多个其它服务器充当从服务器。
  MySQL复制基于主服务器在二进制日志中跟踪所有对数据库的更改(插入、更新、删除等等)，必须在主服务器上启用二进制日志。
  MySQL使用3个线程来执行复制同步功能，其中两个线程(Sql线程和IO线程)在从服务器，另外一个线程(IO线程)在主服务器。
  当从服务器发出start slave服务时，从服务器创建一个I/O线程，以连接主服务器并让它发送记录在其二进制日志中的语句。
  主服务器创建Binlog Dump线程将二进制日志中的内容发送到从服务器。
  从服务器I/O线程读取主服务器Binlog Dump线程发送的内容并将该数据拷贝到从服务器数据目录中的本地文件中(中继日志),接收到的日志内容依次写入到 Slave 端的Relay Log文件(mysql-relay-bin.xxxxxx)的最末端，并将读取到的Master端的bin-log的文件名和位置记录到master- info文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log的哪个位置开始往后的日志内容，请发给我”。SQL线程读取中继日志并执行日志中包含的更新。在从服务器上，读取和执行更新语句被分成两个独立的任务。
  当从服务器启动时，其I/O线程可以很快地从主服务器索取所有二进制日志内容，然后执行sql线程。

- Mysql 5.6 特性

  由于Mysql 5.6 引入了 GTID(Global Transaction ID)，保证 Slave 在复制的时候不会重复执行相同的事务操作；其次，是用全局事务 IDs 代替由文件名和物理偏移量组成的复制位点，定位 Slave 需要复制的 binlog 内容，在旧的 binlog 事件基础上新增两类事件：
  1.Previous_gtids_log_event 该事件之前的全局事务 ID 集合
  2.Gtid_log_event 标记之后的事务对应的全局事务 ID
  MySQL 5.6 的 binlog 文件中，每个事务的开始不是“BEGIN”，而是 Gtid_log_event 事件。
  binary log
  详解可以参考 https://gitsea.com/wp-content/uploads/2013/06/MySQL_Innovation_Day_Replication_HA.pdf

  - 

  - 优点:
    - 1.使用 GTIDs 作为主备复制的位点，在写 binlog 时用 Gtid_log_event 标记事务
    - 2.主从复制不再基于master的binary logfile和logfile postition,从服务器连接到主服务器之后，把自己曾经获取到的GTID(Retrieved_Gtid_Set)发给主服务器，主服务器把从服务器缺少的GTID及对应的transactions发过去即可.
    - 3.采用多个sql线程，每个sql线程处理不同的database，提高了并发性能，即使某database的某条语句暂时卡住，也不会影响到后续对其它的database进行操作.

- 配置Master-Master
  - 一.创建同步用户

    分别在两台mysql上执行
    mysql> grant replication slave on *.* to 'replicate'@'%' identified by '123456';
    mysql> flush privileges;
  - 二.修改my.cnf

    修改 /etc/my.cnf 前最好做个 ...
    - Master-A 的my.cnf配置
      - binlog-format=ROW //二进制日志的格式，有row、statement和mixed几种类型,
      - log-slave-updates=true
      - gtid-mode=on //启用GTID
      - enforce-gtid-consistency=true //启用GTID
      - master-info-repository=TABLE//默认是file，选择table方式保存
      - relay-log-info-repository=TABLE//默认是file，选择table方式保存
      - sync-master-info=1 //实时同步
      - slave-parallel-workers=2 //设定从服务器的SQL线程数；0表示关闭多线程复制功能
      - binlog-checksum=CRC32 //日志校验
      - master-verify-checksum=1//启用校验
      - slave-sql-verify-checksum=1//启用校验
      - binlog-rows-query-log-events=1//只对row-based binlog有效
      - server-id=1
      - report-port=3307
      - port=3306
      - log-bin=master-a-bin.log
      - report-host=192.168.1.168
      - innodb_flush_log_at_trx_commit=1 //)每N次事务提交或事务外的指令都需要把日志写入（flush）硬盘

- sync_binlog=1 //This makes MySQL synchronize the binary log’s contents to disk each time it commits a transaction
- auto_increment_offset = 1// 设置AUTO_INCREMENT起点
- auto_increment_increment = 2//设置AUTO_INCREMENT增量
- replicate-do-db = test//需要同步的数据库
- replicate-ignore-db = mysql,information_schema,performance_schema//不需要同步的数据库

- Master-B 的my.cnf配置
  - binlog-format=ROW
  - log-slave-updates=true
  - gtid-mode=on
  - enforce-gtid-consistency=true
  - master-info-repository=TABLE
  - relay-log-info-repository=TABLE
  - sync-master-info=1
  - slave-parallel-workers=2
  - binlog-checksum=CRC32
  - master-verify-checksum=1
  - slave-sql-verify-checksum=1
  - binlog-rows-query-log-events=1
  - server-id=2
  - report-port=3307
  - port=3306
  - log-bin=master-a-bin.log
  - report-host=192.168.1.169
  - innodb_flush_log_at_trx_commit=1
  - sync_binlog=1
  - auto_increment_offset = 1
  - auto_increment_increment = 2
  - replicate-do-db = test
  - replicate-ignore-db = mysql,information_schema,performance_schema

- 三.重启Mysql，启动Slave服务
  - 首先重启两台mysql服务，
  - 在Master-A 执行如下操作
    - change master to master_host='192.168.1.169', master_user='replicate',master_password='123456',master_auto_position=1;
    - start slave;
  - 在Master-B 执行如下操作
    - change master to master_host='192.168.1.168', master_user='replicate',master_password='123456',master_auto_position=1;
    - start slave;
  - 接下来就可以测试了，两边的test数据库增加不同的数据，都会同步到另外一台服务器上
  - 同时还可以通过 show slave status G;查看相关服务状态
- 配置Keepalived
  - 修改keepalived.cnf 文件，默认放置/etc/keepalived/

    在开启keepalived服务之前先关闭防火墙，keepalived服务会占用112和255端口
    然后通过sudo service keepalived start 开启服务，读者可以随意开启和停止keepalived测试看看IP有没有自动切换。
    在遇到问题的时候可以通过ip add show eth0 和tail -f /var/log/messages进行问题定位。

    - Master-A 的keepalived.conf配置如下
      - ! Configuration File for keepalived
      -
      - bal_defs {
      - notification_email {
      - ****@163.com
      - }
      - notification_email_from ***8@qq.com
      - smtp_server smtp.qq.com
      - smtp_connect_timeout 30

- router_id LVS_DEVEL
- }
-
- vrrp_instance VI_1 {
- state MASTER
- interface eth0 //网卡
- virtual_router_id 51 //同一实例下virtual_router_id必须相同
- priority 100 //定义优先级，数字越大，优先级越高 BACKUP 优先级要低于MASTER
- advert_int 1 //MASTER与BACKUP负载均衡器之间同步检查的时间间隔，单位是秒
- authentication { //验证类型和密码
- auth_type PASS
- auth_pass 1111
- }
- virtual_ipaddress { //VIP 组
- 192.168.1.100
- }
- }
- vrrp_instance VI_2 {
- state BACKUP
- interface eth0
- virtual_router_id 52
- priority 50
- advert_int 1
- authentication {
- auth_type PASS
- auth_pass 1111
- }
- virtual_ipaddress {
- 192.168.1.200
- }
- }

- Master-B keepalived.conf 配置如下
  - ! Configuration File for keepalived
  -
  - bal_defs {
  - notification_email {
  - ***@163.com
  - }
  - notification_email_from ***@qq.com
  - smtp_server smtp.qq.com
  - smtp_connect_timeout 30
  - router_id LVS_DEVEL
  - }
  -
  - vrrp_instance VI_1 {
  - state BACKUP
  - interface eth0
  - virtual_router_id 51
  - priority 50
  - advert_int 1
  - authentication {
  - auth_type PASS
  - auth_pass 1111
  - }

- virtual_ipaddress {
- 192.168.1.100
- }
- }
- vrrp_instance VI_2 {
- state MASTER
- interface eth0
- virtual_router_id 52
- priority 100
- advert_int 1
- authentication {
- auth_type PASS
- auth_pass 1111
- }
- virtual_ipaddress {
- 192.168.1.200
- }
- }

- 锦上添花

添加mysql监控功能，当一台服务器的mysql进程挂掉之后，自动重启mysql服务，如果重启失效，则停止运行keepalived，进行容灾切换
首先安装nmap，运行yum -y install nmap
分别在两台服务器上的/opt目录下新增chk_mysql.sh脚本，内容如下
#!/bin/sh
# check mysql server status
PORT=3306
nmap localhost -p $PORT | grep "$PORT/tcp open"
#echo $?
if [$? -ne 0];then
service mysql stop
service mysql start
sleep 5
nmap localhost -p $PORT | grep "$PORT/tcp open"
[$? -ne 0] && service keepalived stop
fi
增加可执行权限 chmod +x /opt/chk_mysql.sh
把Master-A的keepalived.conf 修改成如下内容
! Configuration File for keepalived
bal_defs {
notification_email {
****@163.com
}
notification_email_from ***8@qq.com
smtp_server smtp.qq.com
smtp_connect_timeout 30
router_id LVS_DEVEL
}
vrrp_script chk_mysql_port {
script "/opt/chk_mysql.sh"
interval 2
weight 2
}
vrrp_instance VI_1 {
state BACKUP
interface eth0 //网卡
virtual_router_id 51 //同一实例下virtual_router_id必须相同
priority 50 //定义优先级，数字越大，优先级越高 BACKUP 优先级要低于MASTER
advert_int 1 //MASTER与BACKUP负载均衡器之间同步检查的时间间隔，单位是秒
authentication { //验证类型和密码
auth_type PASS
auth_pass 1111
}
track_script {
chk_mysql_port
}
virtual_ipaddress { //VIP 组
192.168.1.100
}
}
vrrp_instance VI_2 {
state MASTER
interface eth0
virtual_router_id 52
priority 100
advert_int 1
authentication {
auth_type PASS
auth_pass 1111
}
track_script {
chk_mysql_port
}
virtual_ipaddress {
192.168.1.200
}
}
同理，相应的修改Master-B的配置文件。
重启服务生效之后，你会发现，手动关闭mysql服务之后，会被keepalived自动开启服务。

- 3. 主动-被动模式的Master-Master (Master-Master in Active-Passive Mode)
  - 特点:
    - 这是master-master结构变化而来的，它避免了M-M的缺点，实际上，这是一种具有容错和高可用性的系统
    - 它的不同点在于其中一个服务只能进行只读操作
- 4. 带 从服务器的Master-Master结构(Master-Master with Slaves)
  - 这种结构的优点就是提供了冗余。在地理上分布的复制结构，它不存在单一节点故障问题
  - 而且还可以将读密集型的请求放到slave上

- 性能优化 =>
  - **MySQL 分表:**

分区和分表的比较:
传统分表后，count、sum等统计操作只能对所有切分表进行操作后之后在应用层再次计算得出最后统计数据。而分区表则不受影响，可直接统计。
1. 分区对原系统改动最小，分区只涉及数据库层面，应用层不需要做出改动。
2. 分区有个限制是主表的所有唯一字段（包括主键）必须包含分区字段，而分表没有这个限制。
3. 分表包括垂直切分和水平切分，而分区只能起到水平切分的作用。

    - 基础知识:

- 分区管理
  - 判断当前MySQL是否支持分区?
    - mysql> show variables like '%partition%';
    - +------------------+-------+
    - | **Variable_name** | **Value** |
    - +------------------+-------+
    - | **have_partitioning** | **YES** |
    - +------------------+-------+
    - **1 row in set (0.00 sec)**
  - 1. 增加分区:
    - 对于Range分区和LIst分区来说:
      - alter table table_name add partition (partition p0 values ...(exp))
      - values后面的内容根据分区的类型不同而不同。
    - 对于Hash分区和Key分区来说:
      - alter table table_name add partition partitions 8;
      - 上面的语句，指的是新增8个分区 。
  - 2. 删除分区
    - 对于Range分区和List分区:
      - alter table table_name drop partition p0; //p0为要删除的分区名称
      - 删除了分区，同时也将删除该分区中的所有数据。同时，如果删除了分区导致分区不能覆盖所有值，那么插入数据的时候会报错。
    - 对于Hash和Key分区:
      - alter table table_name coalesce partition 2; //将分区缩减到2个
      - coalesce [ˌkəʊə'les] vi. 联合，合并
  - 3. 分区查询
    - 1）查询某张表一共有多少个分区

      information_schema.p ...
      - mysql> select
      - -> partition_name,
      - -> partition_expression,
      - -> partition_description,
      - -> table_rows
      - -> from
      - -> INFORMATION_SCHEMA.partitions
      - -> where
      - -> table_schema='test'
      - -> and table_name = 'emp';
      - +---------------+----------------------+----------------------+------------+
      - | partition_name | partition_expression | partition_description | table_rows |
      - +---------------+----------------------+----------------------+------------+
      - | p0 | store_id | 10 | 0 |
      - | p1 | store_id | 20 | 1 |
      - +---------------+----------------------+----------------------+------------+
    - 2）查看执行计划，判断查询数据是否进行了分区过滤

      partitions:p1 表示数据在p ...
      - mysql> explain partitions select * from emp where store_id=10 \G;
      - *************************** 1. row ***************************
      - id: 1
      - select_type: SIMPLE
      - table: emp
      - partitions: p1
      - type: system
      - possible_keys: NULL
      - key: NULL

- key_len: NULL
- ref: NULL
- rows: 1
- Extra:
- 1 row in set (0.00 sec)

- 表分区基础知识：
  - 1. 什么是表分区？

    表分区，是指根据一定规则，将数据库中的一张表分解成多个更小的，容易管理的部分。从逻辑上看，只有一张表，但是底层却是由多个物理分区组成。
  - 2. 表分区与分表的区别

    分表：指的是通过一定规则，将一张表分解成多张不同的表。比如将用户订单记录根据时间成多个表。
    分表与分区的区别在于：分区从逻辑上来讲只有一张表，而分表则是将一张表分解成多张表。
  - 3. 表分区有什么好处？
    - 1）分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备。
    - 2）和单个磁盘或者文件系统相比，可以存储更多数据
    - 3）优化查询。在where语句中包含分区条件时，可以只扫描一个或多个分区表来提高查询效率；涉及sum和count语句时，也可以在多个分区上并行处理，最后汇总结果。
    - 4）分区表更容易维护。例如：想批量删除大量数据可以清除整个分区。
    - 5）可以使用分区表来避免某些特殊的瓶颈，例如InnoDB的单个索引的互斥访问，ext3问价你系统的inode锁竞争等
  - 4. 分区表的限制因素
    - 1）一个表最多只能有1024个分区
    - 2） MySQL5.1中，分区表达式必须是整数，或者返回整数的表达式。在MySQL5.5中提供了非整数表达式分区的支持。
    - 3）如果分区字段中有主键或者唯一索引的列，那么多有主键列和唯一索引列都必须包含进来。即：分区字段要么不包含主键或者索引列，要么包含全部主键和索引列。
    - 4）分区表中无法使用外键约束
    - 5）MySQL的分区适用于一个表的所有数据和索引，不能只对表数据分区而不对索引分区，也不能只对索引分区而不对表分区，也不能只对表的一部分数据分区。
- 垂直分区（纵向分区）-- 宽表

  分布式存储（业务）

  什么是纵向分区呢？就是竖向分区了，举例来说明，在设计用户表的时候，开始的时候没有考虑好，而把个人的所有信息都放到了一张表里面去，这样这个表里面就会有比较大的字段，如个人简介，而这些简介呢，也许不会有好多人去看，所以等到有人要看的时候，在去查找，分表的时候，可以把这样的大字段，分开来。
- **水平分区的几种模式：**

  MySQL 分区表有四种类型

  MySQL 作为开源数据库已经支持分区表，而 PostgreSQL 在分区表方面本身做得不是很好，虽然可以通过继承实现表分区功能。接下来打算了解 MySQL 分区表相关的内容。
  - RANGE 分区（范围）

    
    - 按照数据的区间范围分区，区间要连续并且不能互相重叠
    - Range 分区的使用 & 示例：
      - 一. 查看是否支持分区
        - 方法 1 - 查看帮助信息（服务器端）

          查看一下，如果发现有上面这个东西，说明他是支持分区的，默认是打开的。如果你已经安装过了mysql的话
          - [root@BlackGhost mysql-5.1.50]# ./configure --help |grep -A 3 Partition
          - === Partition Support ===
          - Plugin Name: partition
          - Description: MySQL Partitioning Support
          - Supports build: static
        - 方法 2 - 通过 "show plugins;" 命令 & SQL 查看：
          - root@localhost:(none)>use information_schema
          -
          - select plugin_name,plugin_version,plugin_status ,plugin_type
          - from plugins where plugin_type='STORAGE ENGINE';
        - 方法 3 - 通过 variables 查看：

          查看一下变量，如果支持的话，会有上面的提示的
          - mysql> show variables like "%part%";
          - +-------------------+-------+
          - | Variable_name | Value |
          - +-------------------+-------+

- | have_partitioning | YES |
- +------------------+-------+
- 1 row in set (0.00 sec)

- 备注:
  - 如果以上结果没有 partition 为 active 的信息，则当前 MySQL 数据库不支持分区。

- 二. RANGE 分区举例: 通过 RANGE COLUMNS 分区方式
  - --建表
    - create table tbl_access_log (
    - id bigint,
    - name varchar(64),
    - create_time datetime
    -)
    - PARTITION BY RANGE COLUMNS (create_time) (
    - PARTITION p201410 values less than ('2014-11-01 00:00:00'),
    - PARTITION p201411 values less than ('2014-12-01 00:00:00'),
    - PARTITION p201412 values less than ('2015-01-01 00:00:00'),
    - PARTITION pmax values less than maxvalue
    -);
  - --查看分区
    - root@localhost:information_schema>select table_schema,table_name,partition_name,PARTITION_METHOD from information_schema.partitions where table_name='tbl_access_log';
    - +--------------+---------------+----------------+-----------------+
    - | table_schema | table_name | partition_name | PARTITION_METHOD |
    - +--------------+---------------+----------------+-----------------+
    - | francs | tbl_access_log | p201410 | RANGE COLUMNS |
    - | francs | tbl_access_log | p201411 | RANGE COLUMNS |
    - | francs | tbl_access_log | p201412 | RANGE COLUMNS |
    - | francs | tbl_access_log | pmax | RANGE COLUMNS |
    - +--------------+---------------+----------------+-----------------+
    - 4 rows in set (0.01 sec)
  - --插入数据测试
    - root@localhost:francs>insert into tbl_access_log values(1,'a','2014-10-01 12:00:00');
    - Query OK, 1 row affected (0.06 sec)
    -
    - root@localhost:francs>insert into tbl_access_log values(1,'a','2014-10-01 13:00:00');
    - Query OK, 1 row affected (0.06 sec)
    -
    - root@localhost:francs>insert into tbl_access_log values(1,'a','2014-11-01 13:00:00');
    - Query OK, 1 row affected (0.00 sec)
    -
    - root@localhost:francs>insert into tbl_access_log values(1,'a','2015-01-01 13:00:00');
    - Query OK, 1 row affected (0.00 sec)
    -
    - root@localhost:francs>select * from tbl_access_log;
    - +------+------+---------------------+
    - | id | name | create_time |
    - +------+------+---------------------+
    - | 1 | a | 2014-10-01 12:00:00 |
    - | 1 | a | 2014-10-01 13:00:00 |
    - | 1 | a | 2014-11-01 13:00:00 |
    - | 1 | a | 2015-01-01 13:00:00 |
    - +------+------+---------------------+
    - 4 rows in set (0.00 sec)
  - --查询指定分区

- root@localhost:francs>select * from tbl_access_log PARTITION (p201410);
- +------+------+---------------------+
- | id | name | create_time |
- +------+------+---------------------+
- | 1 | a | 2014-10-01 12:00:00 |
- | 1 | a | 2014-10-01 13:00:00 |
- +------+------+---------------------+
- 2 rows in set (0.00 sec)
-
- root@localhost:francs>select * from tbl_access_log PARTITION (p201411);
- +------+------+---------------------+
- | id | name | create_time |
- +------+------+---------------------+
- | 1 | a | 2014-11-01 13:00:00 |
- +------+------+---------------------+
- 1 row in set (0.00 sec)
-
- root@localhost:francs>select * from tbl_access_log PARTITION (pmax);
- +------+------+---------------------+
- | id | name | create_time |
- +------+------+---------------------+
- | 1 | a | 2015-01-01 13:00:00 |
- +------+------+---------------------+
- 1 row in set (0.00 sec)

- 三. RANGE 分区举例: 通过 RANGE 分区方式
  - --建表
    - create table tbl_access_log_unix (
    - id bigint,
    - name varchar(64),
    - create_time TIMESTAMP
    -)
    - PARTITION BY RANGE (UNIX_TIMESTAMP(create_time)) (
    - PARTITION p201410 values less than (UNIX_TIMESTAMP('2014-11-01 00:00:00')),
    - PARTITION p201411 values less than (UNIX_TIMESTAMP('2014-12-01 00:00:00')),
    - PARTITION p201412 values less than (UNIX_TIMESTAMP('2015-01-01 00:00:00')),
    - PARTITION pmax values less than (maxvalue)
    -);
  - --查看分区类型
    - root@localhost:information_schema>select table_schema,table_name,partition_name,PARTITION_METHOD from partitions where table_name='tbl_access_log_unix';
    - +--------------+--------------------+----------------+-----------------+
    - | table_schema | table_name | partition_name | PARTITION_METHOD |
    - +--------------+--------------------+----------------+-----------------+
    - | francs | tbl_access_log_unix | p201410 | RANGE |
    - | francs | tbl_access_log_unix | p201411 | RANGE |
    - | francs | tbl_access_log_unix | p201412 | RANGE |
    - | francs | tbl_access_log_unix | pmax | RANGE |
    - +--------------+--------------------+----------------+-----------------+
    - 4 rows in set (0.02 sec)
- 备注:
  - 以上介绍了两种 RANGE 分区例子。第一种是 RANGE COLUMNS partitioning 方式
  - 第二种是 RANGE partitioning 方式，两种方式不同，以后再介绍
  - **每个分区都是按顺序定义的，从最低到最高**

每个分区都是按顺序定义的，从最低到最高。上面的语句，如果将less than(10) 和less than (20)的顺序颠倒过来，那么将报错，如下：
ERROR 1493 (HY000): VALUES LESS THAN value must be strictly increasing for each partition

- RANGE分区存在的问题
  - **1. range范围覆盖问题：当插入的记录中对应的分区键的值不在分区定义的范围中的时候，插入语句会失败**

    上面的例子，如果我插入一条store_id = 30的记录会怎么样呢?
    我们上面分区的时候，最大值是20，如果插入一条超过20的记录，会报错:
    mysql> insert into emp(id,store_id) values(2,30);
    ERROR 1526 (HY000): Table has no partition for value 30
    提示30这个值没有对应的分区。
    - Range 分区定义：

      右边的语句创建了emp表，并根据store_id字段进行分区，小于10的值存在分区p0中，大于等于10，小于20的值存在分区p1中
      - mysql> create table emp(
      - -> id INT NOT null,
      - -> store_id int not null
      - ->)
      - -> partition by range(store_id)(
      - -> partition p0 values less than(10),
      - -> partition p1 values less than(20)
      - ->);
    - 解决办法
      - A. 预估分区键的值，及时新增分区。
      - B. 设置分区的时候，使用values less than maxvalue 子句,MAXVALUE表示最大的可能的整数值。
      - C. 尽量选择能够全部覆盖的字段作为分区键，比如一年的十二个月等。
  - **2. Range分区中，分区键的值如果是NULL，将被作为一个最小值来处理。**
- LIST 分区（预定义列表）：

  - 按照List中的值分区，与RANGE的区别是，range分区的区间范围值是连续的；
    List分区是建立离散的值列表告诉数据库特定的值属于哪个分区
  - List 分区语法 & 使用：
    - 语法：
      - partition by list(exp)(//exp为列名或者表达式
      - partition p0 values in (3,5) //值为3和5的在p0分区
      -)
    - 与Range不同的是，list分区不必生命任何特定的顺序：
      - mysql> create table emp1(
      - -> id int not null,
      - -> store_id int not null
      - ->)
      - -> partition by list(store_id)(
      - **-> partition p0 values in (3,5),**
      - **-> partition p1 values in (2,6,7,9)**
      - ->);
  - List 分区的查询：
    - 准备工作：
      - --创建表
        - create table user_info (
        - user_id int8 primary key,
        - user_name varchar(64),
        - create_time timestamp
        -)
        - PARTITION BY LIST (mod(user_id,4)) (
        - PARTITION user_info_p0 VALUES IN (0),
        - PARTITION user_info_p1 VALUES IN (1),
        - PARTITION user_info_p2 VALUES IN (2),
        - PARTITION user_info_p3 VALUES IN (3)
        -);
      - --批量插入数据存储过程

- DELIMITER //
- CREATE PROCEDURE pro_ins_user_info()
- BEGIN
- DECLARE i INT;
- SET i = 1;
-
- WHILE i <= 10000 DO
- insert into user_info(user_id,user_name,create_time)
  values(i,CONCAT('user_',i),current_timestamp);
- SET i = i + 1;
- END WHILE;
- END
- //
- DELIMITER ;

- --执行函数
  - root@localhost:francs>call pro_ins_user_info() ;
  - Query OK, 1 row affected (17.19 sec)

- 查看分区信息：
  - --查看分区表数据分布情况
    - root@localhost:francs> select table_schema,table_name,partition_name,PARTITION_METHOD
      from information_schema.partitions where table_name='user_info';
    - +--------------+------------+----------------+-----------------+
    - | table_schema | table_name | partition_name | PARTITION_METHOD |
    - +--------------+------------+----------------+-----------------+
    - | francs | user_info | user_info_p0 | LIST |
    - | francs | user_info | user_info_p1 | LIST |
    - | francs | user_info | user_info_p2 | LIST |
    - | francs | user_info | user_info_p3 | LIST |
    - +--------------+------------+----------------+-----------------+
    - 4 rows in set (0.07 sec)
  - --查看各分区数据
    - root@localhost:francs>select count(*) from user_info PARTITION (user_info_p0);
    - +----------+
    - | count(*) |
    - +----------+
    - | 2500 |
    - +----------+
    - 1 row in set (0.01 sec)
    -
    - root@localhost:francs>select count(*) from user_info PARTITION (user_info_p1);
    - +----------+
    - | count(*) |
    - +----------+
    - | 2500 |
    - +----------+
    - 1 row in set (0.00 sec)
    -
    - root@localhost:francs>select count(*) from user_info PARTITION (user_info_p2);
    - +----------+
    - | count(*) |
    - +----------+
    - | 2500 |
    - +----------+
    - 1 row in set (0.00 sec)
    -

- root@localhost:francs>select count(*) from user_info PARTITION (user_info_p3);
- +----------+
- | count(*) |
- +----------+
- | 2500 |
- +----------+
- 1 row in set (0.01 sec)

- -- 查看各分区数据 - 2：
  - root@localhost:francs>select * from user_info PARTITION (user_info_p0) limit 3;
  - +---------+-----------+---------------------+
  - | user_id | user_name | create_time |
  - +---------+-----------+---------------------+
  - | 4 | user_4 | 2014-12-03 13:38:11 |
  - | 8 | user_8 | 2014-12-03 13:38:11 |
  - | 12 | user_12 | 2014-12-03 13:38:11 |
  - +---------+-----------+---------------------+
  - 3 rows in set (0.00 sec)
  -
  - root@localhost:francs>select * from user_info PARTITION (user_info_p1) limit 3;
  - +---------+-----------+---------------------+
  - | user_id | user_name | create_time |
  - +---------+-----------+---------------------+
  - | 1 | user_1 | 2014-12-03 13:38:11 |
  - | 5 | user_5 | 2014-12-03 13:38:11 |
  - | 9 | user_9 | 2014-12-03 13:38:11 |
  - +---------+-----------+---------------------+
  - 3 rows in set (0.00 sec)
  -
  - root@localhost:francs>select * from user_info PARTITION (user_info_p2) limit 3;
  - +---------+-----------+---------------------+
  - | user_id | user_name | create_time |
  - +---------+-----------+---------------------+
  - | 2 | user_2 | 2014-12-03 13:38:11 |
  - | 6 | user_6 | 2014-12-03 13:38:11 |
  - | 10 | user_10 | 2014-12-03 13:38:11 |
  - +---------+-----------+---------------------+
  - 3 rows in set (0.00 sec)
  -
  - root@localhost:francs>select * from user_info PARTITION (user_info_p3) limit 3;
  - +---------+-----------+---------------------+
  - | user_id | user_name | create_time |
  - +---------+-----------+---------------------+
  - | 3 | user_3 | 2014-12-03 13:38:11 |
  - | 7 | user_7 | 2014-12-03 13:38:11 |
  - | 11 | user_11 | 2014-12-03 13:38:11 |
  - +---------+-----------+---------------------+
  - 3 rows in set (0.00 sec)
- --查看执行计划
  - 根据分区键 user_idd 查询
    - francs@localhost:francs>explain partitions select * from user_info where user_id=1\G
    - *************************** 1. row ***************************
    - id: 1
    - select_type: SIMPLE
    - table: user_info

- partitions: user_info_p1
- type: const
- possible_keys: PRIMARY
- key: PRIMARY
- key_len: 8
- ref: const
- rows: 1
- Extra: NULL
- 1 row in set (0.06 sec)

- 根据非分区键 user_name 查询
  - francs@localhost:francs>explain partitions select * from user_info where user_name='user_1'\G
  - *************************** 1. row ***************************
  - id: 1
  - select_type: SIMPLE
  - table: user_info
  - partitions: user_info_p0,user_info_p1,user_info_p2,user_info_p3
  - type: ALL
  - possible_keys: NULL
  - key: NULL
  - key_len: NULL
  - ref: NULL
  - rows: 10000
  - Extra: Using where
  - 1 row in set (0.00 sec)
- 备注：
  - **根据分区键查询走了分区 user_info_p1，而根据非分区键查询走了所有分区**

- 注意事项：
  - **如果插入的记录对应的分区键的值不在list分区指定的值中，将会插入失败。并且，list不能像range分区那样提供maxvalue**

- **HASH （哈希）分区：**
  - 目的：
    - Hash分区主要用来分散热点读，确保数据在预先确定个数的分区中尽可能平均分布。
  - 分类（MySQL支持两种Hash分区）：
    - **常规 Hash 分区**
      - 使用取模算法
        - 语法：
          - partition by hash(store_id) partitions 4;
        - 上面的语句，根据store_id对4取模，决定记录存储位置
        - 比如store_id = 234的记录，MOD(234,4)=2,所以会被存储在第二个分区
      - **优点：**
        - 能够使数据尽可能的均匀分布
      - **缺点：**
        - 不适合分区经常变动的需求
          - 假如我要新增加两个分区，现在有6个分区，那么MOD(234,6)的结果与之前MOD(234,4)的结果就会出现不一致，这样大部分数据就需要重新计算分区
          - 为解决此问题，MySQL提供了线性Hash分区
    - **线性 Hash 分区**
      - 分区函数是一个线性的2的幂的运算法则
        - 语法：
          - partition by LINER hash(store_id) partitions 4;

            与常规Hash的不同在于，“Liner” ...
        - 算法介绍:

假设要保存记录的分区编号为N,num为一个非负整数,表示分割成的分区的数量，那么N可以通过以下步骤得到：
Step 1. 找到一个大于等于num的2的幂，这个值为V，V可以通过下面公式得到：
V = Power(2,Ceiling(Log(2,num)))
例如：刚才设置了4个分区，num=4，Log(2,4)=2,Ceiling(2)=2,power(2,2)=4,即V=4
Step 2. 设置N=F(column_list)&(V-1)
例如：刚才V=4，store_id=234对应的N值，N = 234&（4-1） =2
Step 3. 当N>=num,设置V=Ceiling(V/2),N=N&(V-1)
例如：store_id=234,N=2
假设上面算出来的N=5，那么V=Ceiling(2.5)=3,N=234&(3-1)=1,即在第一个分区。

- **优点:**
  - 在分区维护（增加，删除，合并，拆分分区）时，MySQL能够处理得更加迅速
- **缺点:**
  - 与常规Hash分区相比，线性Hash各个分区之间的数据分布不太均衡

- KEY（键值） 分区:
  - 目的:
    - 类似Hash分区，Hash分区允许使用用户自定义的表达式，但Key分区不允许使用用户自定义的表达式
    - Hash仅支持整数分区，而Key分区支持除了Blob和text的其他类型的列作为分区键
  - 语法:
    - partition by key(exp) partitions 4;//exp是零个或多个字段名的列表
    - key分区的时候，exp可以为空，如果为空，则默认使用主键作为分区键，没有主键的时候，会选择非空惟一键作为分区键
- Composite（复合模式 / 子区分）:
  - 目的:
    - 分区表中对每个分区再次分割，又称为复合分区。

- 分区表的使用及查询优化

  - 根据实际情况选择分区方法

对现有表分区的原则与传统分表一样。
传统的按照增量区间分表对应于分区的Range分区，比如对表的访问多是近期产生的新数据，历史数据访问较少，则可以按一定时间段（比如年或月）或一定数量（比如100万）对表分区，具体根据哪种取决于表索引结构。分区后最后一个分区即为近期产生的数据，当一段时间过后数据量再次变大，可对最后一个分区重新分区（REORGANIZE PARTITION）把一段时间（一年或一月）或一定数量（比如100万）的数据分离出去。
传统的散列方法分表对应于分区的Hash / Key分区，具体方法上面已经介绍过。

  - 查询优化
    - 用explain partitions命令来查看SQL对于分区的使用情况

一般来说，就是在where条件中加入分区 ...

      - mysql> show create table salaries\G;
      - *************************** 1. row ***************************
      - Table: salaries
      - Create Table: CREATE TABLE `salaries` (
      - `emp_no` int(11) NOT NULL,
      - `salary` int(11) NOT NULL,
      - `from_date` date NOT NULL,
      - `to_date` date NOT NULL,
      - PRIMARY KEY (`emp_no`,`from_date`)
      -) ENGINE=InnoDB DEFAULT CHARSET=utf8
      - /*!50100 PARTITION BY RANGE (year(from_date))
      - (PARTITION p1 VALUES LESS THAN (1985) ENGINE = InnoDB,
      - PARTITION p2 VALUES LESS THAN (1986) ENGINE = InnoDB,
      - PARTITION p3 VALUES LESS THAN (1987) ENGINE = InnoDB,
      - PARTITION p4 VALUES LESS THAN (1988) ENGINE = InnoDB,
      - PARTITION p5 VALUES LESS THAN (1989) ENGINE = InnoDB,
      - PARTITION p6 VALUES LESS THAN (1990) ENGINE = InnoDB,
      - PARTITION p7 VALUES LESS THAN (1991) ENGINE = InnoDB,
      - PARTITION p8 VALUES LESS THAN (1992) ENGINE = InnoDB,
      - PARTITION p9 VALUES LESS THAN (1993) ENGINE = InnoDB,
      - PARTITION p10 VALUES LESS THAN (1994) ENGINE = InnoDB,
      - PARTITION p11 VALUES LESS THAN (1995) ENGINE = InnoDB,
      - PARTITION p12 VALUES LESS THAN (1996) ENGINE = InnoDB,
      - PARTITION p13 VALUES LESS THAN (1997) ENGINE = InnoDB,
      - PARTITION p14 VALUES LESS THAN (1998) ENGINE = InnoDB,

- PARTITION p15 VALUES LESS THAN (1999) ENGINE = InnoDB,
- PARTITION p16 VALUES LESS THAN (2000) ENGINE = InnoDB,
- PARTITION p17 VALUES LESS THAN (2001) ENGINE = InnoDB,
- PARTITION p18 VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */
- 下面的查询没有利用分区，因为partitions中包含了所有的分区：
  - mysql> explain partitions select * from salaries where salary > 100000\G;
  - *************************** 1. row ***************************
  - id: 1
  - select_type: SIMPLE
  - table: salaries
  - partitions: p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18
  - type: ALL
  - possible_keys: NULL
  - key: NULL
  - key_len: NULL
  - ref: NULL
  - rows: 2835486
  - Extra: Using where
- 只有在where条件中加入分区列才能起到作用，过滤掉不需要的分区：
  - mysql> explain partitions select * from salaries where salary > 100000 and from_date > '1998-01-01'\G;
  - *************************** 1. row ***************************
  - id: 1
  - select_type: SIMPLE
  - table: salaries
  - partitions: p15,p16,p17,p18
  - type: ALL
  - possible_keys: NULL
  - key: NULL
  - key_len: NULL
  - ref: NULL
  - rows: 1152556
  - Extra: Using where
- 与普通搜索一样，在运算符左侧使用函数将使分区过滤失效，即使与分区函数想同也一样：
  - mysql> explain partitions select * from salaries where salary > 100000 and year(from_date) > 1998\G;
  - *************************** 1. row ***************************
  - id: 1
  - select_type: SIMPLE
  - table: salaries
  - partitions: p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18
  - type: ALL
  - possible_keys: NULL
  - key: NULL
  - key_len: NULL
  - ref: NULL
  - rows: 2835486
  - Extra: Using where

- 常见的性能优化方式：

  - 1、选取最适用的字段属性
    - 在可能的情况下，应该尽量把字段设置为NOT NULL，这样在将来执行查询的时候，数据库不用去比较NULL值
  - 2、使用连接（JOIN）来代替子查询(Sub-Queries)
  - 3、使用联合(UNION)来代替手动创建的临时表
  - 4、事务

- 介绍：

```
BEGIN;
INSERT INTO salesinfo SET customerid=14;
UPDATE inventory SET quantity =11 WHERE item='book';
COMMIT;
```

  - 事物以BEGIN关键字开始，COMMIT关键字结束。在这之间的一条SQL操作失败，那么，ROLLBACK命令就可以把数据库恢复到BEGIN开始之前的状态。
- 优点：
  - 事务是维护数据库完整性的一个非常好的方法
- 缺点：

由于在事务执行的过程中，数据库将会被锁定，因此其它的用户请求只能暂时等待直到该事务结束。如果一个数据库系统只有少数几个用户来使用，事务造成的影响不会成为一个太大的问题；但假设有成千上万的用户同时访问一个数据库系统，例如访问一个电子商务网站，就会产生比较严重的响应延迟。

  - 因为它的独占性，有时会影响数据库的性能，尤其是在很大的应用系统中
  - 由于在事务执行的过程中，数据库将会被锁定，因此其它的用户请求只能暂时等待直到该事务结束
  - 例如访问一个电子商务网站，就会产生比较严重的响应延迟。

- 5、锁定表
  - 可以维护数据的完整性
- 6、使用外键
  - 保证数据的关联性
- 7、使用索引

索引是提高数据库性能的常用方法，它可以令数据库服务器以比没有索引快得多的速度检索特定的行，尤其是在查询语句当中包含有MAX(),MIN()和ORDERBY这些命令的时候，性能提高更为明显

  - 索引应建立在那些将用于JOIN,WHERE判断和ORDERBY排序的字段上
  - 尽量不要对数据库中某个含有大量重复的值的字段建立索引
  - 对于一个ENUM类型（枚举类型）的字段来说，出现大量重复值是很有可能的情况
- 8、优化的查询语句
  - a、 首先，最好是在相同类型的字段间进行比较的操作

在MySQL3.23版之前，这甚至是一个 ...

例如不能将一个建有索引的INT字段和BIGINT字段进行比较；但是作为特殊的情况，在CHAR类型的字段和VARCHAR类型字段的字段大小相同的时候，可以将它们进行比较。

  - b、 其次，在建有索引的字段上尽量不要使用函数进行操作

使用YEAE()函数时，将会使索引不能发 ...

  - c、第三，在搜索字符型字段时，我们有时会使用LIKE关键字和通配符，这种做法虽然简单，但却也是以牺牲系统性能为代价的

```
例如下面的查询将会比较表中的每一条记录。
SELECT * FROM books WHERE name like "MySQL%"
    但是如果换用下面的查询，返回的结果一样，但速度就要快上很多：
SELECT * FROM books WHERE name > = "MySQL" and name < "MySQM"
```

- MySQL 去重操作优化实战：
  - 思路 1 ：巧用索引和变量
    - 问题背景：

元旦假期收到 阿里吴老师 来电，被告知已将MySQL查重SQL优化到极致：100万原始数据，其中50万重复，把去重后的50万数据写入目标表只需要9秒钟。这是一个惊人的数字，要知道仅是insert 50万条记录也需要些时间的。于是来了兴趣，自己实验、思考、总结做了一遍。

    - 二、实验环境
      - Linux虚机：CentOS release 6.4；8G内存；
      - 100G机械硬盘；双物理CPU双核，共四个处理器；MySQL 5.6.14
    - 一、问题提出
      - 源表t_source结构如下：
        - item_id int,
        - created_time datetime,
        - modified_time datetime,
        - item_name varchar(20),
        - other varchar(20)
      - 1. 源表中有100万条数据，其中有50万created_time和item_name重复。
      - 2. 要把去重后的50万数据写入到目标表。
      - 3. 重复created_time和item_name的多条数据，可以保留任意一条，不做规则限制。
    - 三、建立测试表和数据
      - 1. 建立源表
        - create table t_source

- (
- item_id int,
- created_time datetime,
- modified_time datetime,
- item_name varchar(20),
- other varchar(20)
-);

- 2. 建立目标表
  - create table t_target like t_source;
- 3. 生成100万测试数据，其中有50万created_time和item_name重复
  - create procedure sp_generate_data()
  - begin
  - set @i := 1;
  -
  - while @i<=500000 do
  - set @created_time := date_add('2017-01-01',interval @i second);
  - set @modified_time := @created_time;
  - set @item_name := concat('a',@i);
  - insert into t_source
  - values (@i,@created_time,@modified_time,@item_name,'other');
  - set @i:=@i+1;
  - end while;
  - commit;
  -
  - set @last_insert_id := 500000;
  - insert into t_source
  - select item_id + @last_insert_id,
  - created_time,
  - date_add(modified_time,interval @last_insert_id second),
  - item_name,
  - 'other'
  - from t_source;
  - commit;
  - end
  - //
  - delimiter ;
  -
  - call sp_generate_data();
- 源表没有主键或唯一性约束，有可能存在两条完全一样的数据，所以再插入一条记录模拟这种情况。
  - insert into t_source
  - select * from t_source where item_id=1;
  - commit;
- 查询总记录数和去重后的记录数图一所示
  - select count(*),count(distinct created_time,item_name) from t_source;
  -

```
mysql> select count(*),count(distinct created_time,item_name) from t_source;
+----------+----------------------------------------+
| count(*) | count(distinct created_time,item_name) |
+----------+----------------------------------------+
|  1000001 |                                 500000 |
+----------+----------------------------------------+
1 row in set (2.84 sec)
```


可以看到，源表中有1000001条记录，...

- 四、无索引对比测试
  - 1. 使用相关子查询
    - truncate t_target;

- insert into t_target
- select distinct t1.* from t_source t1 where item_id in
- (select min(item_id) from t_source t2 where t1.created_time=t2.created_time and t1.item_name=t2.item_name);
- commit;
-

```
mysql> explain select distinct t1.* from t_source t1 where item_id in
    -> (select min(item_id) from t_source t2 where t1.created_time=t2.created_time and t1.item_name=t2.item_name);
+----+--------------------+-------+------+---------------+------+---------+------+--------+-----------------------------+
| id | select_type        | table | type | possible_keys | key  | key_len | ref  | rows   | Extra                       |
+----+--------------------+-------+------+---------------+------+---------+------+--------+-----------------------------+
|  1 | PRIMARY            | t1    | ALL  | NULL          | NULL | NULL    | NULL | 996406 | Using where; Using temporary |
|  2 | DEPENDENT SUBQUERY | t2    | ALL  | NULL          | NULL | NULL    | NULL | 996406 | Using where                 |
+----+--------------------+-------+------+---------------+------+---------+------+--------+-----------------------------+
2 rows in set (0.00 sec)
```


这个语句很长时间都出不来结果，只看一下执 ...

- 2. 使用表连接查重
  - truncate t_target;
  - insert into t_target
  - select distinct t1.* from t_source t1,
  - (select min(item_id) item_id,created_time,item_name from t_source group by created_time,item_name) t2
  - where t1.item_id = t2.item_id;
  - commit;
  -

```
mysql> explain select distinct t1.* from t_source t1,
    -> (select min(item_id) item_id,created_time,item_name from t_source group by created_time,item_name) t2
    -> where t1.item_id = t2.item_id;
+----+-------------+------------+------+---------------+-------------+---------+-------------------+--------+-----------------------------+
| id | select_type | table      | type | possible_keys | key         | key_len | ref               | rows   | Extra                       |
+----+-------------+------------+------+---------------+-------------+---------+-------------------+--------+-----------------------------+
|  1 | PRIMARY     | t1         | ALL  | NULL          | NULL        | NULL    | NULL              | 996406 | Using where; Using temporary |
|  1 | PRIMARY     | <derived2> | ref  | <auto_key0>   | <auto_key0> | 5       | test.t1.item_id   |     10 | Distinct                    |
|  2 | DERIVED     | t_source   | ALL  | NULL          | NULL        | NULL    | NULL              | 996406 | Using temporary; Using filesort |
+----+-------------+------------+------+---------------+-------------+---------+-------------------+--------+-----------------------------+
3 rows in set (0.00 sec)
```


这种方法用时35秒，查询计划如图三所示

  - （1）内层查询扫描t_source表的100万行，建立临时表，并使用文件排序找出去重后的最小item_id，生成导出表derived2，此导出表有50万行。
  - （2）MySQL会在临时表derived2上自动创建一个item_id字段的索引auto_key0。
  - （3）外层查询也要扫描t_source表的100万行数据，在与临时表做链接时，对t_source表每行的item_id，使用auto_key0索引查找临时表中匹配的行，并在此时优化distinct操作，在找到第一个匹配的行后即停止查找同样值的动作
- 3. 使用变量
  - set @a:='0000-00-00 00:00:00';
  - set @b:=' ';
  - set @f:=0;
  - truncate t_target;
  - insert into t_target
  - select item_id,created_time,modified_time,item_name,other
  - from
  - (select t0.*,if(@a=created_time and @b=item_name,@f:=0,@f:=1) f, @a:=created_time,@b:=item_name
  - from
  - (select * from t_source order by created_time,item_name) t0) t1 where f=1;
  - commit;
  -

```
mysql> explain select item_id,created_time,modified_time,item_name,other
    ->   from
    -> (select t0.*,if(@a=created_time and @b=item_name,@f:=0,@f:=1) f, @a:=created_time,@b:=item_name
    ->   from
    -> (select * from t_source order by created_time,item_name) t0) t1 where f=1;
+----+-------------+------------+------+---------------+-------------+---------+-------+--------+-----------------+
| id | select_type | table      | type | possible_keys | key         | key_len | ref   | rows   | Extra           |
+----+-------------+------------+------+---------------+-------------+---------+-------+--------+-----------------+
|  1 | PRIMARY     | <derived2> | ref  | <auto_key0>   | <auto_key0> | 4       | const |     10 | NULL            |
|  2 | DERIVED     | <derived3> | ALL  | NULL          | NULL        | NULL    | NULL  | 996406 | NULL            |
|  3 | DERIVED     | t_source   | ALL  | NULL          | NULL        | NULL    | NULL  | 996406 | Using filesort |
+----+-------------+------------+------+---------------+-------------+---------+-------+--------+-----------------+
3 rows in set (0.00 sec)
```


这种方法用时14秒，查询计划如图四所示

  - （1）最内层的查询扫描t_source表的100万行，并使用文件排序，生成导出表derived3。
  - （2）第二层查询要扫描derived3的100万行，生成导出表derived2，完成变量的比较和赋值，并自动创建一个导出列f上的索引auto_key0。

- （3）最外层使用auto_key0索引扫描derived2得到去重的结果行。
- 与方法2比较，变量方法消除了表关联，查询速度提高了2.7倍

- 结论：
  - 至此，我们还没有在源表上创建任何索引。无论使用哪种写法，要查重都需要对created_time和item_name字段进行排序，因此很自然地想到，如果在这两个字段上建立联合索引，可以用于消除filesort，从而提高查询性能

- 五、建立created_time 和 item_name上的联合索引对比测试
  - 1. 建立created_time和item_name字段的联合索引
    - create index idx_sort on t_source(created_time,item_name,item_id);
    - analyze table t_source;
  - 2. 使用相关子查询
    - truncate t_target;
    - insert into t_target
    - select distinct t1.* from t_source t1 where item_id in
    - (select min(item_id) from t_source t2 where t1.created_time=t2.created_time and t1.item_name=t2.item_name);
    - commit;
    -

```
mysql> explain select distinct t1.* from t_source t1 where item_id in
    -> (select min(item_id) from t_source t2 where t1.created_time=t2.created_time and t1.item_name=t2.item_name);
+----+--------------------+-------+-------+---------------+----------+---------+-------------------------------------------+--------+--------------------------+
| id | select_type        | table | type  | possible_keys | key      | key_len | ref                                       | rows   | Extra                    |
+----+--------------------+-------+-------+---------------+----------+---------+-------------------------------------------+--------+--------------------------+
|  1 | PRIMARY            | t1    | ALL   | NULL          | NULL     | NULL    | NULL                                      | 996405 | Using where; Using temporary |
|  2 | DEPENDENT SUBQUERY | t2    | ref   | idx_sort      | idx_sort | 69      | test.t1.created_time,test.t1.item_name |      1 | Using index              |
+----+--------------------+-------+-------+---------------+----------+---------+-------------------------------------------+--------+--------------------------+
2 rows in set (0.00 sec)
```


这种方法用时20秒，查询计划如图五所示

      - （1）外层查询的t_source表是驱动表，需要扫描100万行。
      - （2）对于驱动表每行的item_id，通过idx_sort索引查询出一行数据
  - 3. 使用表连接查重
    - truncate t_target;
    - insert into t_target
    - select distinct t1.* from t_source t1,
    - (select min(item_id) item_id,created_time,item_name from t_source group by created_time,item_name) t2
    - where t1.item_id = t2.item_id;
    - commit;
    -

```
mysql> explain select distinct t1.* from t_source t1,
    -> (select min(item_id) item_id,created_time,item_name from t_source group by created_time,item_name) t2
    -> where t1.item_id = t2.item_id;
+----+-------------+------------+--------+---------------+-------------+---------+----------------+--------+--------------------------+
| id | select_type | table      | type   | possible_keys | key         | key_len | ref            | rows   | Extra                    |
+----+-------------+------------+--------+---------------+-------------+---------+----------------+--------+--------------------------+
|  1 | PRIMARY     | t1         | ALL    | NULL          | NULL        | NULL    | NULL           | 996405 | Using where; Using temporary |
|  1 | PRIMARY     | <derived2> | ref    | <auto_key0>   | <auto_key0> | 5       | test.t1.item_id |     10 | Distinct                 |
|  2 | DERIVED     | t_source   | index  | idx_sort      | idx_sort    | 74      | NULL           | 996405 | Using index              |
+----+-------------+------------+--------+---------------+-------------+---------+----------------+--------+--------------------------+
3 rows in set (0.00 sec)
```


这种方法用时25秒，查询计划如图六所示；...

  - 4. 使用变量
    - set @a:='0000-00-00 00:00:00';
    - set @b:=' ';
    - set @f:=0;
    - truncate t_target;
    - insert into t_target
    - select item_id,created_time,modified_time,item_name,other
    - from
    - (select t0.*,if(@a=created_time and @b=item_name,@f:=0,@f:=1) f,
      @a:=created_time,@b:=item_name
    - from
    - (select * from t_source order by created_time,item_name) t0) t1 where f=1;
    - commit;
    - 这种方法用时14秒，查询计划与没有索引时的相同，如图四所示。可见索引对这种写法没有作用。能不能消除嵌套，只用一层查询出结果呢?
  - 5. 使用变量，并且消除嵌套查询
    - set @a:='0000-00-00 00:00:00';
    - set @b:=' ';

- truncate t_target;
- insert into t_target
- select * from t_source force index (idx_sort)
- where (@a!=created_time or @b!=item_name) and (@a:=created_time) is not null and (@b:=item_name) is not null
- order by created_time,item_name;
- commit;
-

```
mysql> explain select * from t_source force index (idx_sort)
    ->  where (@a!=created_time or @b!=item_name) and (@a:=created_time) is not null and (@b:=item_name) is not null
    ->  order by created_time,item_name;
+----+-------------+----------+-------+---------------+----------+---------+------+--------+-------------+
| id | select_type | table    | type  | possible_keys | key      | key_len | ref  | rows   | Extra       |
+----+-------------+----------+-------+---------------+----------+---------+------+--------+-------------+
|  1 | SIMPLE      | t_source | index | NULL          | idx_sort | 74      | NULL | 996405 | Using where |
+----+-------------+----------+-------+---------------+----------+---------+------+--------+-------------+
1 row in set (0.00 sec)
```


这种方法用时8秒，查询计划如图七所示

- 六、总结 （ 看似一个简单的部分查重语句，要想完美优化，也必须清晰理解很多知识点）

基础要扎实，应用要灵活，方能书写出高效的 ...

- 查询语句的逻辑执行顺序
- 使用索引优化排序
- 强制按索引顺序扫描表
- 索引覆盖
- 半连接查询优化
- 布尔表达式等

- 思路 2 ： 多线程并行执行

- 问题背景：

上一篇已经将单条查重语句调整到最优，但该语句是以单线程方式执行。能否利用多处理器，让去重操作多线程并行执行，从而进一步提高速度呢？比如我的实验环境是4处理器，如果使用4个线程同时执行查重sql，理论上应该接近4倍的性能提升。

- 一、数据分片
  - 分片概念：
    - 我们生成测试数据时，created_time采用每条记录加一秒的方式，也就是最大和在最小的时间差为50万秒，而且数据均匀分布。因此先把数据平均分成4份
  - 1. 查询出4份数据的created_time边界值
    - select date_add('2017-01-01',interval 125000 second) dt1,
    - date_add('2017-01-01',interval 2*125000 second) dt2,
    - date_add('2017-01-01',interval 3*125000 second) dt3,
    - max(created_time) dt4
    - from t_source;
  - 2. 查看每份数据的记录数，确认数据平均分布
    - select case when created_time >= '2017-01-01'
    - and created_time < '2017-01-02 10:43:20'
    - then '2017-01-01'
    - when created_time >= '2017-01-02 10:43:20'
    - and created_time < '2017-01-03 21:26:40'
    - then '2017-01-02 10:43:20'
    - when created_time >= '2017-01-03 21:26:40'
    - and created_time < '2017-01-05 08:10:00'
    - then '2017-01-03 21:26:40'
    - else '2017-01-05 08:10:00'
    - end min_dt,
    - case when created_time >= '2017-01-01'
    - and created_time < '2017-01-02 10:43:20'
    - then '2017-01-02 10:43:20'
    - when created_time >= '2017-01-02 10:43:20'
    - and created_time < '2017-01-03 21:26:40'
    - then '2017-01-03 21:26:40'
    - when created_time >= '2017-01-03 21:26:40'

- and created_time < '2017-01-05 08:10:00'
- then '2017-01-05 08:10:00'
- else '2017-01-06 18:53:20'
- end max_dt,
- count(*)
- from t_source
- group by case when created_time >= '2017-01-01'
- and created_time < '2017-01-02 10:43:20'
- then '2017-01-01'
- when created_time >= '2017-01-02 10:43:20'
- and created_time < '2017-01-03 21:26:40'
- then '2017-01-02 10:43:20'
- when created_time >= '2017-01-03 21:26:40'
- and created_time < '2017-01-05 08:10:00'
- then '2017-01-03 21:26:40'
- else '2017-01-05 08:10:00'
- end,
- case when created_time >= '2017-01-01'
- and created_time < '2017-01-02 10:43:20'
- then '2017-01-02 10:43:20'
- when created_time >= '2017-01-02 10:43:20'
- and created_time < '2017-01-03 21:26:40'
- then '2017-01-03 21:26:40'
- when created_time >= '2017-01-03 21:26:40'
- and created_time < '2017-01-05 08:10:00'
- then '2017-01-05 08:10:00'
- else '2017-01-06 18:53:20'
- end;

- 3. 建立查重的存储过程

有了以上信息我们就可以写出4条语句处理全部数据。为了调用接口尽量简单，建立下面的存储过程

- delimiter //
- create procedure sp_unique(i smallint)
- begin
- set @a:='0000-00-00 00:00:00';
- set @b:=' ';
- if (i<4) then
- insert into t_target
- select * from t_source force index (idx_sort)
- where created_time >= date_add('2017-01-01',interval (i-1)*125000 second)
- and created_time < date_add('2017-01-01',interval i*125000 second)
- and (@a!=created_time or @b!=item_name)
- and (@a:=created_time) is not null
- and (@b:=item_name) is not null
- order by created_time,item_name;
- commit;
- else
- insert into t_target
- select * from t_source force index (idx_sort)
- where created_time >= date_add('2017-01-01',interval (i-1)*125000 second)
- and created_time <= date_add('2017-01-01',interval i*125000 second)
- and (@a!=created_time or @b!=item_name)
- and (@a:=created_time) is not null
- and (@b:=item_name) is not null
- order by created_time,item_name;

- commit;
- end if;
- end
- //
-
- delimiter ;

- 二、并行执行

下面分别使用shell后台进程和MySQ ...

- 1. shell后台进程
  - （1）建立duplicate_removal.sh文件，内容如下
    - #!/bin/bash
    - mysql -vvv -u root -p123456 test -e "truncate t_target" &>/dev/null
    - date '+%H:%M.%N'
    - for y in {1..4}
    - do
    - sql="call sp_unique($y)"
    - mysql -vvv -u root -p123456 test -e "$sql" &>par_sql1_$y.log &
    - done
    - wait
    - date '+%H:%M.%N'
  - （2）执行脚本文件
    - chmod 755 duplicate_removal.sh
    - ./duplicate_removal.sh
  - 总结：
    - 可以看到，每个过程的执行时间均不到3.4秒，因为是并行执行，总的过程执行时间也小于3.4秒，比单线程sql速度提高了近3倍
- 2. MySQL Schedule Event
  - 背景：
    - 吴老师也用到了并行，但他是利用MySQL自带的Schedule Event功能实现的，代码应该和下面的类似
  - （1）建立事件历史日志表
    - -- 用于查看事件执行时间等信息
    - create table t_event_history (
    - dbname varchar(128) not null default '',
    - eventname varchar(128) not null default '',
    - starttime datetime(3) not null default '0000-00-00 00:00:00',
    - endtime datetime(3) default null,
    - issuccess int(11) default null,
    - duration int(11) default null,
    - errormessage varchar(512) default null,
    - randno int(11) default null
    -);
  - （2）修改event_scheduler参数
    - set global event_scheduler = 1;
  - （3）为每个并发线程创建一个事件
    - delimiter //
    - create event ev1 on schedule at current_timestamp + interval 1 hour on completion preserve disable do
    - begin
    - declare r_code char(5) default '00000';
    - declare r_msg text;
    - declare v_error integer;
    - declare v_starttime datetime default now(3);
    - declare v_randno integer default floor(rand()*100001);
    -
    - insert into t_event_history (dbname,eventname,starttime,randno)

- #作业名
- values(database(),'ev1', v_starttime,v_randno);
-
- begin
- #异常处理段
- declare continue handler for sqlexception
- begin
- set v_error = 1;
- get diagnostics condition 1 r_code = returned_sqlstate , r_msg = message_text;
- end;
-
- #此处为实际调用的用户程序过程
- call sp_unique(1);
- end;
-
- update t_event_history set endtime=now(3),issuccess=isnull(v_error),duration=timestampdiff(microsecond,starttime,now(3)), errormessage=concat('error=',r_code,', message=',r_msg),randno=null where starttime=v_starttime and randno=v_randno;
-
- end
- //
-
- create event ev2 on schedule at current_timestamp + interval 1 hour on completion preserve disable do
- begin
- declare r_code char(5) default '00000';
- declare r_msg text;
- declare v_error integer;
- declare v_starttime datetime default now(3);
- declare v_randno integer default floor(rand()*100001);
-
- insert into t_event_history (dbname,eventname,starttime,randno)
- #作业名
- values(database(),'ev2', v_starttime,v_randno);
-
- begin
- #异常处理段
- declare continue handler for sqlexception
- begin
- set v_error = 1;
- get diagnostics condition 1 r_code = returned_sqlstate , r_msg = message_text;
- end;
-
- #此处为实际调用的用户程序过程
- call sp_unique(2);
- end;
-
- update t_event_history set endtime=now(3),issuccess=isnull(v_error),duration=timestampdiff(microsecond,starttime,now(3)), errormessage=concat('error=',r_code,', message=',r_msg),randno=null where starttime=v_starttime and randno=v_randno;
-
- end
- //
-
- create event ev3 on schedule at current_timestamp + interval 1 hour on completion preserve

disable do
- begin
- declare r_code char(5) default '00000';
- declare r_msg text;
- declare v_error integer;
- declare v_starttime datetime default now(3);
- declare v_randno integer default floor(rand()*100001);
-
- insert into t_event_history (dbname,eventname,starttime,randno)
- #作业名
- values(database(),'ev3', v_starttime,v_randno);
-
- begin
- #异常处理段
- declare continue handler for sqlexception
- begin
- set v_error = 1;
- get diagnostics condition 1 r_code = returned_sqlstate , r_msg = message_text;
- end;
-
- #此处为实际调用的用户程序过程
- call sp_unique(3);
- end;
-
- update t_event_history set endtime=now(3),issuccess=isnull(v_error),duration=timestampdiff(microsecond,starttime,now(3)), errormessage=concat('error=',r_code,', message=',r_msg),randno=null where starttime=v_starttime and randno=v_randno;
-
- end
- //
-
- create event ev4 on schedule at current_timestamp + interval 1 hour on completion preserve disable do
- begin
- declare r_code char(5) default '00000';
- declare r_msg text;
- declare v_error integer;
- declare v_starttime datetime default now(3);
- declare v_randno integer default floor(rand()*100001);
-
- insert into t_event_history (dbname,eventname,starttime,randno)
- #作业名
- values(database(),'ev4', v_starttime,v_randno);
-
- begin
- #异常处理段
- declare continue handler for sqlexception
- begin
- set v_error = 1;
- get diagnostics condition 1 r_code = returned_sqlstate , r_msg = message_text;
- end;
-
- #此处为实际调用的用户程序过程
- call sp_unique(4);
- end;

-
- update t_event_history set endtime=now(3),issuccess=isnull(v_error),duration=timestampdiff(microsecond,starttime,now(3)), errormessage=concat('error=',r_code,', message=',r_msg),randno=null where starttime=v_starttime and randno=v_randno;
-
- end
- //
-
- delimiter ;

- （4）触发事件执行
  - mysql -vvv -u root -p123456 test -e "truncate t_target;alter event ev1 on schedule at current_timestamp enable;alter event ev2 on schedule at current_timestamp enable;alter event ev3 on schedule at current_timestamp enable;alter event ev4 on schedule at current_timestamp enable;"
  - 说明：
    - 该命令行顺序触发了4个事件，但不会等前一个执行完才执行下一个，而是立即向下执行。从图六的输出也可以清楚地看到这一点。因此四次过程调用是并行执行的。
  -

```
[root@cdh3~]#mysql -vvv -u root -prockey test -e "alter event ev1 on schedule at current_timestamp enable;alter event ev2 on schedule at current_time
stamp enable;alter event ev3 on schedule at current_timestamp enable;alter event ev4 on schedule at current_timestamp enable;"
Warning: Using a password on the command line interface can be insecure.
--------------
alter event ev1 on schedule at current_timestamp enable
--------------

Query OK, 0 rows affected (0.00 sec)

--------------
alter event ev2 on schedule at current_timestamp enable
--------------

Query OK, 0 rows affected (0.00 sec)

--------------
alter event ev3 on schedule at current_timestamp enable
--------------

Query OK, 0 rows affected (0.00 sec)

--------------
alter event ev4 on schedule at current_timestamp enable
--------------

Query OK, 0 rows affected (0.00 sec)

Bye
```


- （5）查看事件执行日志
  - select * from t_event_history;
  -

```
mysql> select * from t_event_history;
+--------+-----------+-------------------------+-------------------------+-----------+----------+--------------+--------+
| dbname | eventname | starttime               | endtime                 | issuccess | duration | errormessage | randno |
+--------+-----------+-------------------------+-------------------------+-----------+----------+--------------+--------+
| test   | ev1       | 2017-01-09 16:06:48.000 | 2017-01-09 16:06:51.532 |         1 |  3532000 | NULL         |   NULL |
| test   | ev2       | 2017-01-09 16:06:48.000 | 2017-01-09 16:06:51.502 |         1 |  3502000 | NULL         |   NULL |
| test   | ev4       | 2017-01-09 16:06:48.000 | 2017-01-09 16:06:51.532 |         1 |  3532000 | NULL         |   NULL |
| test   | ev3       | 2017-01-09 16:06:48.000 | 2017-01-09 16:06:51.532 |         1 |  3532000 | NULL         |   NULL |
+--------+-----------+-------------------------+-------------------------+-----------+----------+--------------+--------+
4 rows in set (0.00 sec)
```

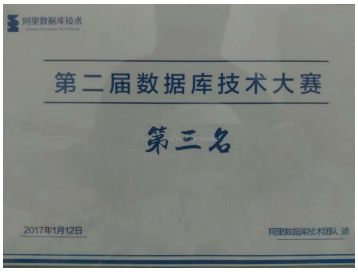

可以看到，每个过程的执行均为3.5秒，又 ...

- 思路 3：用 rocksdb 替代 innodb

- 问题背景：

前面已经建立了索引，优化了SQL语句，并将单线程变为多线程并行执行，去重时间由最初的35秒优化为3.5秒，是不是就到此为止呢？吴老师又使用了rocksdb存储引擎替代innodb的方法。这里有必要交代一下命题的背景。这道MySQL数据库优化的题目出自是阿里内部的竞赛题，当然我是听吴老师口述的，真正的题目及其竞赛规则与竞赛环境不甚明确，但有一条是允许自由选择MySQL存储引擎。在实际的生产环境中，几乎没有可能为了单一操作的优化而改变表的存储引擎，因为这样做的代价通常很高。
顺便提一句，据刚刚得到的消息，吴老师在今天结束的决赛中获得了第三名的好成绩（前两名都是数据库内核组的），祝贺！

-



- 一、MyRocks 简介
  - MyRocks是FaceBook开源的MySQL分支。下面链接是一篇阿里系的介绍文章，在此不再赘述。
- 二、安装配置 MyRocks

1. 支持的平台

- 官方支持OS：
  - CentOS 6.8
  - CentOS 7.2.x
- 经过验证的兼容编译器：
  - gcc 4.8.1
  - gcc 4.9.0
  - gcc 5.4.0
  - gcc 6.1.0
  - Clang 3.9.0
- 2. 安装前准备
  - （1）安装依赖包

    这几个包缺一不可，gflags-deve ...
    - sudo yum install -y cmake gcc-c++ bzip2-devel libaio-devel bison \
    - zlib-devel snappy-devel
    - sudo yum install -y gflags-devel readline-devel ncurses-devel \
    - openssl-devel lz4-devel gdb git
  - （2）安装gflags
    - git clone https://github.com/gflags/gflags.git
    - cd gflags
    - mkdir build && cd build
    - ccmake ..
  - （3）安装lz4
    - git clone https://github.com/lz4/lz4.git
    - cd lz4
    - make
    - make install
  - （4）升级gcc

    系统自带4.4.7，升级到任一验证过的编 ...
    - # 下载源码包并解压
    - wget http://ftp.gnu.org/gnu/gcc/gcc-6.1.0/gcc-6.1.0.tar.bz2
    - tar -jxvf gcc-6.1.0.tar.bz2
    -
    - # 下载编译所需依赖库
    - cd gcc-6.1.0
    - ./contrib/download_prerequisites
    - cd ..
    -
    - # 建立编译输出目录
    - mkdir gcc-build-6.1.0
    -
    - # 进入此目录，执行以下命令，生成makefile文件
    - cd gcc-build-6.1.0
    - ../gcc-6.1.0/configure --enable-checking=release --enable-languages=c,c++ --disable-multilib
    -
    - # 编译，j后面的是核心数，编译速度会比较快
    - make -j4
    -
    - # 安装
    - make install
    -
    - # 切换GCC到新版
    - # 确定新安装的GCC的路径,一般默认在/usr/local/bin下
    - ls /usr/local/bin | grep gcc
    - # 添加新GCC到可选项，倒数第三个是名字，倒数第二个参数为新GCC路径，最后一个参数40为优先级，设

大一些之后就自动使用新版了
- update-alternatives --install /usr/bin/gcc gcc /usr/local/bin/x86_64-pc-linux-gnu-gcc-6.1.0 40
-
- # 定义环境变量，需要设置，否则在编译MyRocks时会报错
- CC='/usr/local/bin/x86_64-pc-linux-gnu-gcc-6.1.0'; export CC;
-
- # 确认当前版本已经切换为新版
- gcc -v

- （5）更新libstdc++
  - cp /root/gcc-build-6.1.0/x86_64-pc-linux-gnu/libstdc++-v3/src/.libs/libstdc++.so.6.0.22 /usr/lib64/
  - cd /usr/lib64/
  - rm -r libstdc++.so.6
  - ln -s libstdc++.so.6.0.22 libstdc++.so.6
  - strings /usr/lib64/libstdc++.so.6 | grep GLIBCXX

- 3. 编译MyRocks源码
  - # 下载项目源码
  - git clone https://github.com/facebook/mysql-5.6.git
  - cd mysql-5.6
  - git submodule init
  - # 调用 git submodule update 用来更新 submodule 信息。
  - git submodule update
  -
  - # 生成makefile文件
  - cmake . -DCMAKE_BUILD_TYPE=RelWithDebInfo -DWITH_SSL=system \
  - -DWITH_ZLIB=bundled -DMYSQL_MAINTAINER_MODE=0 -DENABLED_LOCAL_INFILE=1
  -
  - # 编译
  - make -j8
  -
  - # 安装
  - make install
- 4. 运行mtr tests测试
  - cd mysql-test
  - ./mysql-test-run.pl --mem --async-client --parallel=16 --fast \
  - --max-test-fail=1000 --retry=0 --force --mysqld=--rocksdb \
  - --mysqld=--default-storage-engine=rocksdb --mysqld=--skip-innodb \
  - --mysqld=--default-tmp-storage-engine=MyISAM --suite=rocksdb

- 三、启动 MyRocks
  - 1. 配置 my.cnf
    - [mysqld]
    - # 指定存储引擎
    - rocksdb
    - default-storage-engine=rocksdb
    - # 禁用innodb
    - skip-innodb
    - # 指定临时表使用的存储引擎
    - default-tmp-storage-engine=MyISAM
    - # 指定字符集排序规则
    - collation-server=latin1_bin
    - # 禁用binlog
    - skip_log_bin
    - # 禁用自动提交
    - autocommit=0
    -

- # 不做严格的排序规则检查，如果不设置，在varchar字段建立索引时会报错
- rocksdb_strict_collation_check=0
- # 启用批量数据装载
- rocksdb_bulk_load=1
- # 指定批次记录数
- rocksdb_bulk_load_size=10000
- # 允许并行写内存表
- rocksdb_allow_concurrent_memtable_write = 1
- # 设置写缓冲区大小
- rocksdb_db_write_buffer_size = 1G
- # 不立即刷新内存缓冲区中的数据
- rocksdb_write_disable_wal = 1

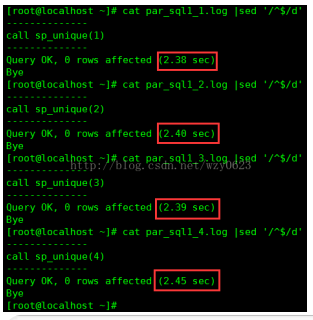
- 2. 使用 mysql_install_db 初始化数据库
  - /usr/local/mysql/scripts/mysql_install_db --defaults-file=/usr/local/mysql/my.cnf --datadir=/usr/local/mysql/data --basedir=/usr/local/mysql
- 3. 启动mysqld
  - useradd mysql
  - chown -R mysql:mysql /usr/local/mysql/data
  - /usr/local/mysql/bin/mysqld_safe --defaults-file=/usr/local/mysql/my.cnf --datadir=/usr/local/mysql/data --basedir=/usr/local/mysql &

- 四、在 MyRocks 上进行去重测试
  - 1. 建立源表
    - create table t_source
    - (
    - item_id int,
    - created_time datetime,
    - modified_time datetime,
    - item_name varchar(20),
    - other varchar(20)
    -);
  - 2. 建立目标表
    - create table t_target like t_source;
  - 3. 建立生成测试数据的存储过程
    - delimiter //
    - create procedure sp_generate_data()
    - begin
    - set @i := 1;
    -
    - while @i<=500000 do
    - set @created_time := date_add('2017-01-01',interval @i second);
    - set @modified_time := @created_time;
    - set @item_name := concat('a',@i);
    - insert into t_source
    - values (@i,@created_time,@modified_time,@item_name,'other');
    - set @i:=@i+1;
    - end while;
    - commit;
    -
    - set @last_insert_id := 500000;
    - insert into t_source
    - select item_id + @last_insert_id,
    - created_time,
    - date_add(modified_time,interval @last_insert_id second),
    - item_name,
    - 'other'

- from t_source;
- commit;
- end
- //
- delimiter ;

- 4. 生成测试数据
  - call sp_generate_data();
  -
  - insert into t_source
  - select * from t_source where item_id=1;
  - commit;
- 5. 建立索引
  - create index idx_sort on t_source(created_time,item_name,item_id);
  - analyze table t_source;
- 6. 建立并行执行的存储过程
  - delimiter //
  - create procedure sp_unique(i smallint)
  - begin
  - set @a:='0000-00-00 00:00:00';
  - set @b:=' ';
  - if (i<4) then
  - insert into t_target
  - select * from t_source force index (idx_sort)
  - where created_time >= date_add('2017-01-01',interval (i-1)*125000 second)
  - and created_time < date_add('2017-01-01',interval i*125000 second)
  - and (@a!=created_time or @b!=item_name)
  - and (@a:=created_time) is not null
  - and (@b:=item_name) is not null
  - order by created_time,item_name;
  - commit;
  - else
  - insert into t_target
  - select * from t_source force index (idx_sort)
  - where created_time >= date_add('2017-01-01',interval (i-1)*125000 second)
  - and created_time <= date_add('2017-01-01',interval i*125000 second)
  - and (@a!=created_time or @b!=item_name)
  - and (@a:=created_time) is not null
  - and (@b:=item_name) is not null
  - order by created_time,item_name;
  - commit;
  - end if;
  - end
  - //
  -
  - delimiter ;
- 7. 建立shell脚本文件duplicate_removal.sh
  - #!/bin/bash
  - /usr/local/mysql/bin/mysql -vvv test -e "truncate t_target" &>/dev/null
  - date '+%H:%M.%N'
  - for y in {1..4}
  - do
  - sql="call sp_unique($y)"
  - /usr/local/mysql/bin/mysql -vvv test -e "$sql" &>par_sql1_$y.log &
  - done

        - wait
        - date '+%H:%M.%N'
      - 8. 执行去重
        - chmod 755 duplicate_removal.sh
        - ./duplicate_removal.sh
    - 并行执行的4个过程调用分别用时
      -



      可以看到，每个过程的执行时间均在2.4秒 ...

- MySQL 对于千万级的大表要怎么优化?

  成本也是由低到高，mysql数据库一般都 ...

  - 1. 优化 SQL & 索引；同时在 Where 条件中尽可能过滤无用数据
  - 2. 使用缓存：Memcached & Redis
  - **3. 主从复制、主主复制、读写分离**
    - 可以在应用层做，效率高，也可以用三方工具，第三方工具推荐360的atlas,其它的要么效率不高，要么没人维护
  - 4. 利用 MySLQ 自带的分区表：（可省略）

    第四如果以上都做了还是慢，不要想着去做切分，mysql自带分区表，先试试这个，对你的应用是透明的，无需更改代码,但是sql语句是需要针对分区表做优化的，sql条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，另外分区表还有一些坑，在这里就不多说了；
    - 尽可能在 Where 条件中过滤掉无用的数据，将全表扫描，优化为扫描局部数据
  - **5. 垂直拆分**
    - 根据业务的耦合度，将一根大的系统分成多个小的系统，也就是分布式系统
  - 6. 水平切分：

    第六才是水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，**要选择一个合理的sharding key，为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表；**

- **MySQL 存储引擎详解:**

  MySQL有多种存储引擎：
  MyISAM、InnoDB、MERGE、MEMORY(HEAP)、BDB(BerkeleyDB)、EXAMPLE、FEDERATED、ARCHIVE、CSV、BLACKHOLE。
  MySQL支持数个存储引擎作为对不同表的类型的处理器。MySQL存储引擎包括处理事务安全表的引擎和处理非事务安全表的引擎：
  再说一下不同引擎的优化，myisam读的效果好，写的效率差，这和它数据存储格式，索引的指针和锁的策略有关的，它的数据是顺序存储的（innodb数据存储方式是聚簇索引），他的索引btree上的节点是一个指向数据物理位置的指针，所以查找起来很快，（innodb索引节点存的则是数据的主键，所以需要根据主键二次查找）；myisam锁是表锁，只有读读之间是并发的，写写之间和读写之间（读和插入之间是可以并发的，去设置concurrent_insert参数，定期执行表优化操作，更新操作就没有办法了）是串行的，所以写起来慢，并且默认的写优先级比读优先级高，高到写操作来了后，可以马上插入到读操作前面去，如果批量写，会导致读请求饿死，所以要设置读写优先级或设置多少写操作后执行读操作的策略;myisam不要使用查询时间太长的sql，如果策略使用不当，也会导致写饿死，所以尽量去拆分查询效率低的sql,innodb一般都是行锁，这个一般指的是sql用到索引的时候，行锁是加在索引上的，不是加在数据记录上的，如果sql没有用到索引，仍然会锁定表,mysql的读写之间是可以并发的，普通的select是不需要锁的，当查询的记录遇到锁时，用的是一致性的非锁定快照读，也就是根据数据库隔离级别策略，会去读被锁定行的快照，其它更新或加锁读语句用的是当前读，读取原始行；因为普通读与写不冲突，所以innodb不会出现读写饿死的情况，又因为在使用索引的时候用的是行锁，锁的粒度小，竞争相同锁的情况就少，就增加了并发处理，所以并发读写的效率还是很优秀的，问题在于索引查询后的根据主键的二次查找导致效率低；ps:很奇怪，为什innodb的索引叶子节点存的是主键而不是像mysism一样存数据的物理地址指针吗？如果存的是物理地址指针不就不需要二次查找了吗，这也是我开始的疑惑，根据mysism和innodb数据存储方式的差异去想，你就会明白了，我就不费口舌了！所以innodb为了避免二次查找可以使用索引覆盖技术，无法使用索引覆盖的，再延伸一下就是基于索引覆盖实现延迟关联；不知道什么是索引覆盖的，建议你无论如何都要弄清楚它是怎么回事！尽你所能去优化你的sql吧！说它成本低，却又是一项费时费力的活，需要在技术与业务都熟悉的情况下，用心去优化才能做到最优，优化后的效果也是立竿见影的！

  - MySQL 常见的存储引擎:

    互联网项目中**随着硬件成本的降低及缓存、中间件的应用，一般我们选择都以 InnoDB 存储引擎为主，很少再去选择 MyISAM 了。而业务真发展的一定程度时，自带的存储引擎无法满足时，这时公司应该是有实力去自主研发满足自己需求的存储引擎或者购买商用的存储引擎了。**
    - MyISAM
      - **MyISAM 是 mysql 5.5.5 之前的默认引擎，它支持 B-tree/FullText/R-tree 索引类型。**
      - 锁级别为表锁，表锁优点是开销小，加锁快；缺点是锁粒度大，发生锁冲动概率较高，容纳并发能力低，这个引擎适合查询为主的业务。
      - 此引擎不支持事务，也不支持外键。
      - MyISAM强调了快速读取操作。它存储表的行数，于是SELECT COUNT(*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。
      - MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。MyISAM在所有MySQL配置里被支持，它是默认的存储引擎，除非你配置MySQL默认使用另外一个引擎
    - InnoDB
      - **InnoDB 存储引擎最大的亮点就是支持事务，支持回滚，它支持 Hash/B-tree 索引类型。**
      - 锁级别为行锁，行锁优点是适用于高并发的频繁表修改，高并发是性能优于 MyISAM。缺点是系统消耗较大，索引不仅缓

存自身，也缓存数据，相比 MyISAM 需要更大的内存。
    - InnoDB 中不保存表的具体行数，也就是说，执行 select count(*) from table时，InnoDB 要扫描一遍整个表来计算有多少行。
    - 支持事务，支持外键。
  - Memory
    - **Memory 是内存级别存储引擎，数据存储在内存中，所以他能够存储的数据量较小**
    - 因为内存的特性，存储引擎对数据的一致性支持较差。锁级别为表锁，不支持事务。但访问速度非常快，并且默认使用 hash 索引
    - Memory存储引擎使用存在内存中的内容来创建表，每个Memory表只实际对应一个磁盘文件，在磁盘中表现为.frm文件。
    - MEMORY存储引擎提供“内存中”表。MERGE存储引擎允许集合将被处理同样的MyISAM表作为一个单独的表
    - 就像MyISAM一样，MEMORY和MERGE存储引擎处理非事务表，这两个引擎也都被默认包含在MySQL中。
    - MEMORY存储引擎正式地被确定为HEAP引擎
- 其他存储引擎：
  - BDB：
    - 可替代InnoDB的事务引擎，支持COMMIT、ROLLBACK和其他事务特性
  - Merge：
    - 允许MySQL DBA或开发人员将一系列等同的MyISAM表以逻辑方式组合在一起，并作为1个对象引用它们。对于诸如数据仓储等VLDB环境十分适合。
  - Archive：
    - 为大量很少引用的历史、归档、或安全审计信息的存储和检索提供了完美的解决方案
  - Federated：
    - 能够将多个分离的MySQL服务器链接起来，从多个物理服务器创建一个逻辑数据库。十分适合于分布式环境或数据集市环境
  - Cluster/NDB：
    - MySQL的簇式数据库引擎，尤其适合于具有高性能查找要求的应用程序，这类查找需求还要求具有最高的正常工作时间和可用性
  - Other：
    - 其他存储引擎包括CSV（引用由逗号隔开的用作数据库表的文件），Blackhole（用于临时禁止对数据库的应用程序输入），以及Example引擎（可为快速创建定制的插件式存储引擎提供帮助）
  - ◆ EXAMPLE存储引擎
    - 是一个“存根”引擎，它不做什么。你可以用这个引擎创建表，但没有数据被存储于其中或从其中检索
    - 这个引擎的目的是服务，在 MySQL源代码中的一个例子，它演示说明如何开始编写新存储引擎。同样，它的主要兴趣是对开发者。
  - ◆ NDB Cluster
    - 是被MySQL Cluster用来实现分割到多台计算机上的表的存储引擎
    - 它在MySQL-Max 5.1二进制分发版里提供
    - 这个存储引擎当前只被Linux, Solaris, 和Mac OS X 支持。在未来的MySQL分发版中，我们想要添加其它平台对这个引擎的支持，包括Windows
  - ARCHIVE存储引擎
    - 用来无索引地，非常小地覆盖存储的大量数据
  - CSV存储引擎
    - 把数据以逗号分隔的格式存储在文本文件中
  - BLACKHOLE存储引擎
    - 接受但不存储数据，并且检索总是返回一个空集
  - ◆ FEDERATED存储引擎
    - 把数据存在远程数据库中。在MySQL 5.1中，它只和MySQL一起工作，使用MySQL C Client API
    - 在未来的分发版中，我们想要让它使用其它驱动器或客户端连接方法连接到另外的数据源。
- 如何使用：
  - 当你创建一个新表的时候，你可以通过添加一个ENGINE 或TYPE 选项到CREATE TABLE语句来告诉MySQL你要创建什么类型的表：

    虽然TYPE仍然在MySQL 5.1中被支持，现在ENGINE是首选的术语。
    - **CREATE TABLE t (i INT) ENGINE = INNODB;**
    -
    - **CREATE TABLE t (i INT) TYPE = MEMORY;**
  - 查看系统支持的存储引擎：
    - mysql> show engines;
    - +------------+---------+--+
    - | Engine | Support | Comment |

- +------------+---------+--+
- | MyISAM | DEFAULT | Default engine as of MySQL 3.23 with great performance |
- | MEMORY | YES | Hash based, stored in memory, useful for temporary tables |
- | InnoDB | YES | Supports transactions, row-level locking, and foreign keys |
- | BerkeleyDB | NO | Supports transactions and page-level locking |
- | BLACKHOLE | NO | /dev/null storage engine (anything you write to it disappears) |
- | EXAMPLE | NO | Example storage engine |
- | ARCHIVE | NO | Archive storage engine |
- | CSV | NO | CSV storage engine |
- | ndbcluster | NO | Clustered, fault-tolerant, memory-based tables |
- | FEDERATED | NO | Federated MySQL storage engine |
- | MRG_MYISAM | YES | Collection of identical MyISAM tables |
- | ISAM | NO | Obsolete storage engine |
- +------------+---------+--+

- 问题汇总 =>
  - 数据库和缓存一致性问题，出现的场景，解决方案
  - **数据结构 =>**

    介绍下MySQL，B+树，存储引擎
    mysql 索引是什么数据结构, 为什么用b+树，和b树区别?
    mysql 从二叉查找树说到红黑树说到多叉树再到b到b+，最后让手写了一个**二叉查找树的中序遍历。**
    mysql索引为什么用的是b+ tree而不是b tree、红黑树
    MySQL索引的实现，innodb的索引，b+树索引是怎么实现的，为什么用b+树做索引节点，一个节点存了多少数据，怎么规定大小，与磁盘页对应。
    Innodb引擎一个数据页的大小为什么是16KB？20个mysql字段，都是varchar(10),1000w数据情况下b+树高度是多少?
    mysql数据库底层存储
    B树和B+树的区别性?
    为什么使用B+树存储 -- 这里与红黑树进行了比较，然后操作系统文件系统的页相关回答
    mysql为什么用b+树，和b树区别?
    mysql 从二叉查找树说到红黑树说到多叉树再到b到b+，最后让手写了一个二叉查找树的中序遍历。
    - MySQL 支持的索引：
      - B+树索引
      - 全文索引
      - 哈希索引
    - B+ 树和 B 树的区别

      1、B+树的层级更少：相较于B树B+每个非叶子节点存储的关键字数更多，树的层级更少所以查询数据更快；
      2、B+树查询速度更稳定：B+所有关键字数据地址都存在叶子节点上，所以每次查找的次数都相同所以查询速度要比B树更稳定；
      3、B+树天然具备排序功能：B+树所有的叶子节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比B树高。
      4、B+树全节点遍历更快：B+树遍历整棵树只需要遍历所有的叶子节点即可，，而不需要像B树一样需要对每一层进行遍历，这有利于数据库做全表扫描。
      B树相对于B+树的优点是，如果经常访问的数据离根节点很近，而B树的非叶子节点本身存有关键字其数据的地址，所以这种数据检索的时候会要比B+树快。
      - 平衡二叉树、B树、B+树、B*树 的原理 =>

        
    - MySQL 为什么选择使用 B+ 树

      
      - 1. B+树的磁盘读取代价低
      - 2. B+树的查询效率更加稳定
      - 3. B+树便于执行扫库操作
    - B+ 树的数据存储深度 =>

      MySQL索引的实现，innodb的索引，b+树索引是怎么实现的，为什么用b+树做索引节点，一个节点存了多少数据，怎么规定大小，与磁盘页对应。
      Innodb引擎一个数据页的大小为什么是16KB？20个mysql字段，都是varchar(10),1000w数据情况下b+树高度是多少?
      - 基础概念 =>
        - InnoDB一棵B+树可以存放 2000万 行数据

          所以在**InnoDB中B+树高度一般为1-3层，它就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次IO操作即可查找到数据。**
          - innodb_page_size 的大小默认是 16k
            - 16384
          - 非叶子节点能存放多少指针
            - 我们假设主键ID为bigint类型，长度为8字节，而指针大小在InnoDB源码中设置为6字节，这样一共14字节
          - 我们一个页中能存放多少这样的单元，其实就代表有多少指针
            - 即 16384/14=1170
          - 可以算出一棵高度为2的B+树
            - 能存放1170*16=18720条这样的数据记录
          - 一个高度为3的B+树可以存放

                - 1170*1170*16=21902400
            - 如果存储 varchar(10) 的字符 =>
                - 一个页面可以存放的单元：
                    - 16384/16=1024
                - 可以算出一棵高度为2的B+树
                    - 1024 * 16 = 16384
                - 一个高度为3的B+树可以存放
                    - 1024 * 1024 * 16 = 16777216
        - 怎么得到InnoDB主键索引B+树的高度?
        - 为什么一个节点为1页（16k）就够了?

            对着上面Mysql中Innodb中对B+树的实际应用（主要看主键索引），我们可以发现，B+树中的一个节点存储的内容是：
            非叶子节点：主键+指针
            叶子节点：数据
            那么，假设我们一行数据大小为1K，那么一页就能存16条数据，也就是一个叶子节点能存16条数据；再看非叶子节点，假设主键ID为bigint类型，那么长度为8B，指针大小在Innodb源码中为6B，一共就是14B，那么一页里就可以存储16K/14=1170个(主键+指针)，那么一颗高度为2的B+树能存储的数据为：1170*16=18720条，一颗高度为3的B+树能存储的数据为：1170*1170*16=21902400（千万级条）。所以在InnoDB中B+树高度一般为1-3层，它就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次IO操作即可查找到数据。所以也就回答了我们的问题，1页=16k这么设置是比较合适的，是适用大多数的企业的，当然这个值是可以修改的，所以也能根据业务的时间情况进行调整。
- 基础知识：
    - ACID
    - 三大范式
- 锁：
    - 行锁、表锁、间隙锁、意向锁
- 索引：

    聚集索引与非聚集索引的区别
    MySQL索引的实现
    数据库索引的作用是什么
    mysql的聚集索引（回答说是根据主键构建B+树，概念错了）
    myisam索引的组织方式?
    稠密索引是个啥
    - 聚集索引
    - 非聚集索引
    - MySQL索引的实现
        - MySQL 的 B+Tree
        - Myisam 中的 B+Tree
        - InnoDB 中的 B+Tree
- 事务
    - MVCC
        - 内部原理 =>
        - 基本使用
    - MVCC

        MVCC (Multiversion Concurrency Control) 中文全程叫多版本并发控制，是现代数据库（包括 MySQL、Oracle、PostgreSQL 等）引擎实现中常用的处理读写冲突的手段，目的在于提高数据库高并发场景下的吞吐性能。
        - 内部原理 =>
        - 基本使用
    - 事务隔离级别

        Mysql(innondb 下同) 有哪几种事务隔离级别?
        不同事务隔离级别分别会加哪些锁?
        MySQL的事务隔离级别，分别解决什么问题。
        mysql实现事务的原理(MVCC)
- 存储引擎 =>

    innodb索引原理?
    myisam的索引原理?
    为什么myisam支持压缩表?
- 最左(前缀)匹配原则概念 =>

说说什么是最左匹配?
我们模拟数据建立一个联合索引
select *, concat(right(emp_no,1), "-", right(title,1), "-", right(from_date,2)) from employees.titles limit 10;
那么对应的B+树为
https://user-gold-cdn.xitu.io/2019/3/8/1695c7d76e87e04e?imageView2/0/w/1280/h/960/format/webp/ignore-error/1
我们判断一个查询条件能不能用到索引，我们要分析这个查询条件能不能利用某个索引缩小查询范围对于select * from employees.titles where emp_no = 1是能用到索引的，因为它能利用上面的索引所有查询范围，首先和第一个节点“4-r-01”比较，1

- 应用 Demo =>

- 二叉树 -> 索引 -> 优化 -> 引擎