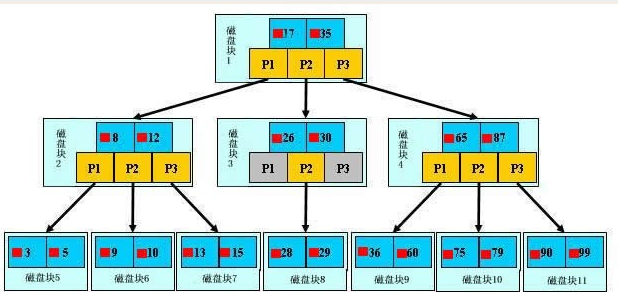
## mysql部分

1. Mysql索引原理？

索引的目的是提高查询效率，本质上是存储引擎用于快速定位记录的一种数据结构。

mysql采用的是B+数数据结构作为索引的存储。

介绍之前需要先知道磁盘IO和预读：磁盘IO是很昂贵的，局部预读性说明当计算机访问一个地址的数据的时候，与其相邻的数据也会被读到内存中。每次IO读取的单元是page一般是4k/8k，这就是一次IO，索引原理中一定要尽可能尽可能低的IO读取次数。



上图是B+数，浅蓝色代表数据块，可以认为是page页（不规范），深蓝色存储的是包含的数据项，黄色是指针，上图中磁盘块1包含数据项17和35，包含指针p1、p2、p3，其中p1表示小于17数据项的磁盘块，p2表示在17和35之间数据的数据块，p3表示大于35的磁盘块。真是数据全部储存于叶子结点3 5 9.....。非叶子节点不存储真是数据，只存储指引搜索方向的数据项，如17和35并不是真是的数据。

B+树查找流程：

如果要查找29，那么首先把磁盘块1加载到内存，此时发生了一次IO，在内存中用二分查找确定29在17-35之间，锁定p2指向的磁盘块3，将磁盘块3加载到内存中，此时发生第二次IO，同样的29在26和30之间，锁定磁盘块8，再发生第三次IO就把数据所有的数据找到了。

B+树性质：

①索引字段尽量要小，因为磁盘块大小一次IO最大是一个page，所以如果字段过大，树的高度越大，IO次数将越多

②索引的最左匹配，比如多个索引（name,age,sex），首先会根据name来决定下一步搜索方向，如果name缺失，将不知道走哪里，将会遍历所有记录了，所以如果有符合索引，查询的时候一个顶要按照顺序加where条件。

最左匹配有个特别之处就是当where中只有=和in的时候可以乱序，mysql自动识别来优化索引。

Mysql索引分类：

①普通索引index：加速查找

②唯一索引：

主键索引：primary key：加速查找+约束（唯一且不为空）

唯一索引：unique：加速查找+约束（唯一）

③联合索引：

-primary key(id, name)：联合主键索引

-unique(id, name)：联合唯一索引

-index(id, name)：联合普通索引

④全文索引full text：用于搜索很长一篇文章时效果好

⑤空间索引spatial：几乎不用

注意区分联合索引和索引合并的区别

索引的类型：

①hash索引

查询单条块，范围查询慢

②btree索引

在这里先说一下为什么不用hash索引吧，两个原因：

第一：业务场景中很多是需要范围查询的，所以如果只查一条那么hash快，但是范围查询那么B+数的叶子结点的链表的能力就显现出来了

第二：有时候索引过大是无法一次性加载到内存中的，而B+树允许一次只加载一个节点。

创建索引原则：

①最左匹配原则，需要注意一点就是当遇到范围查询字段的时候索引匹配结束，所以where后面把范围条件放到最后去

②=和in可以乱序

③尽量选择区分度高的字段做索引count(distinct col1)/count(\*)

④所以列不能参与计算

索引无法命中的情况：

①like ‘%xx’ 左模糊

②索引使用函数

③or：如果or两边都是索引的话没事

④类型不一致，比如string字段查询时不加引号也会不走索引

⑤普通索引的!=和范围查询不会命中索引，特例：字段类型是int的时候会走索引（这里貌似有点问题）

⑥排序字段为索引时select 的字段也需要是索引字段

⑦组合索引最左前缀原则(name, age)的连联合索引，where name=..and agel=...会使用索引，where name = ...会使用，但是where age = ...不使用索引

最后再说一下覆盖索引：

在where条件中用到了索引并且命中了，如果此时select中的字段含有这个索引字段，那么这就叫做覆盖索引，会提升查询速度的！！

慢查询优化的基本步骤：

①先看看是不是真的慢，注意设置SQL\_NO\_CACHE

②where条件单标查，锁定最小返回记录表。意思就是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高

③explain查看执行计划，看是都与②中预期的一样（从锁定记录最少的表开始查询）

④order by limit 形式的sql语句让排序的表优先查

⑤加索引参照上述的索引规则

1. Mysql锁机制？

锁分类：

①按照对数据操作的类型分

共享锁（读锁）：多个读操作互不影响

排它锁（写锁）：当前写操作没有完成之前会阻断其他的写锁和读锁（注意：这里到底是锁哪里呢，是整个表还是某个行要搞清，因为假如只是行锁的话是不影响其他数据的读和写的！）

②按照对数据操作的粒度分

表级锁

行级锁

页级锁

表级锁：偏向MyISAM存储引擎，开销小、加锁快、无死锁；锁定粒度大、发生冲突概率大，并发度低。

行级锁：偏向InnoDB存储引擎，开销大、加锁慢、会出现死锁；锁定粒度小，发生冲突概率低，并发度高。

页面所：开销和加锁时间介于表锁和行锁之间、会出现死锁；锁定粒度也是介于之间。并发度一般。

MyISAM与InnoDB比较：

MyISAM存储引擎由于自身不支持事务，所以用的少，对于锁也相对比较简单：

表级共享锁：不影响其他用户对数据的读取，但是会阻塞其他用户对数据的写！

表级排它锁：阻塞所有对该表的操作！

所以适合读。

InnoDB存储引擎比较复杂，下面进行详细介绍。

InnoDB行锁的实现方式？

InnoDB行锁是通过给索引上的索引项加锁来实现的，只有通过索引条件检索数据，InnoDB才是用行级锁，否则InnoDB将使用表级锁，在实际应用中，要特别主要InnoDB行锁这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。

说一说意向锁的作用吧：

意向锁的作用就是为了解决由于行锁和表锁的共存性而出现的，举个栗子，事务A锁定了一行记录，然后事务B想要锁表，此时事务B要进行两个步骤，首先看是不是有其他事物锁表，如果没有就看有没有行锁存在，因为如果行锁存在肯定不能再上表锁，因为表锁要锁所有记录啊，冲突啊，那么事务B怎么知道有没有行锁，正常来说是全表扫描的！！效率不高，而此时如果有意向锁的话那么就简便多了，如果发现有表的意向共享锁或者意向排它锁，那么事务B上锁的动作就阻塞。注意：：：意向锁是mysql给加上的，不受事务控制。

什么时候用表锁？什么时候用行锁？很显然与索引关系很大，因为行锁是锁索引。

①在不通过索引条件查询的时候，InnoDB确实使用的是表锁，而不是行锁。

②由于InnoDB的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然访问不同行的记录，但是如果是使用相同的索引建，会出现锁冲突。

③当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行，另外，不论是使用主键索引唯一索引、普通索引，InnoDB都会使用行锁来对数据加锁。

④即便在条件中使用了索引字段，但是否使用索引来检索数据是有Mysql通过判断不同执行计划的代价来决定的，如果mysql任务全表扫描相率更高，比如对很小的表，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此在分析锁冲突的时候，别忘了检查sql的执行计划，以确定是否真正使用了索引。

所以：这就是为什么查询语句最好要有where 索引条件！！！这样才会提高效率。

同时表的建立可以没有主键但是索引一定要建立！！！！

什么是间隙锁？

当我们用范围条件而不是相等条件检索数据，并请求共享或者排它锁是，InnoDB会给符合条件的已有数据记录的索引加锁；对于键值在条件范围但并不存在的记录，叫做“间隙GAP”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-key锁）。

间隙锁虽然可以防止幻读，但是也会造成无辜锁定导致其他操作等待。

死锁问题？

当两个事务都需要对方释放资源时发生死锁，InnoDB自带检查机制，会将影响记录少的事务回滚，但是最好设置InnoDB\_lock\_wait\_timeout，这个值不仅适用于死锁，当大量查询并发等待挂起时可以减少压力。

如何避免死锁？

①在应用中，如果不同的程序会并发存储多个表，应尽量约定一相同的顺序来访问表，这样可以大大降低产生死锁的机会。

②在程序以批量方式处理数据的时候，如果实现对数据排序，保证每个线程按固定顺序来处理记录，也可以大大降低死锁的可能。

③在事务中，如果要更新记录，应该直接申请足够的级别锁，即排它锁而不是先申请共享锁，更新时再申请排它锁，因为当用户申请排他锁是，其他事务可能又已经获得相同记录的共享锁。从而造成锁冲突，甚至死锁。

④在repeatable-read隔离级别下，如果两个线程同时对相同条件记录用select for update加排它锁，在没有符合条件记录情况下，两个线程都会加锁成功，程序发现记录不存在就试图插入一条新的记录，如果两个线程都这么做就会出现死锁。这种情况下，将隔离级别改为read-commitred就可以避免

⑤当隔离级别为read-committed时，如果两个线程都先执行了select ... fro update,判断是否存在符合条件的记录，如果没有就插入记录，此时，只有一个线程会插入成功，另一个线程会出现锁等待，当一个线程提交之后，第二个线程会因主键重出错，但索然这个线程出错了却获得一个排它锁，这是如果有第三个线程又来申请排它锁，也会出现死锁。对于这种情况，可以直接做插入操作，然后再捕获主键重异常或者在遇到主键重错误时总是执行rollback释放获得的排它锁。

InnoDB也可以应用表级锁：

①当涉及表大部分数据都需要改变时

②当事务涉及多个表、比较复杂、可能引起死锁时

1. Mysql事务隔离你了解多少？

事务的四大特性（ACID）：

①原子性（Atomicity）：指事务包含的所有操作要么全部成功，要么全部失败回滚

②一致性（Consistency）：指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是一个事务执行前后必须处于一致性状态

③隔离性（Isolation）：当多个用户并发访问数据库时，比如操作同一张表，数据库为每个用户开启事务，不能被其他事务所干扰，多个并发事务要相互隔离

④持久性（Durability）：指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便在数据库系统遇到故障的情况下也不会丢失提交事务的操作

不考虑事务性情况下会发生什么？

①脏读：指一个事务处理过程中读取了另一个未提交的事务中的数据

②不可重复读：指对于数据库中的某个数据，一个事务范围内多次查询结果不同，这是由于查询间隔被另一个事务修改了

③虚读（幻读）：幻读是事务非独立执行时发生的现象。比如事务A整体修改了一批数据，B在A后面新插入一条原始数据，A再次读取的时候发现：咦不对啊，这就是发生了幻觉

总结：脏读是读取了未提交的脏数据，不可重复读和幻读都是读取了另外事务已提交的数据，不同之处就是不可重复读指的是同一个数据项，而幻读针对一批数据体。

Mysql提供的四种隔离级别？

①Serializable：串行化，可以避免所有情形，说白了就是完全阻塞

②Repeatble read：可重复度，避免了脏读、不可重复读

③Read committed：读已提交，可避免脏读

④Read uncommitted：读未提交，任何情况都可能发生

1. 请列举MyISAM和InnoDB引擎的五大区别，越多越好？

①存储结构不同：

MyISAM在磁盘上存储三个文件：.frm(表定义) .MYD(数据) .MYI(索引)

InnoDB在磁盘上存储一个文件，如果比较大就是多个文件

②存储空间不同：

MyISAM可被压缩，存储空间小，支持三种不同的存储格式：静态表、动态表、压缩表

InnoDB需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引

③可移植性、备份及恢复

MyISAM数据以文件的形式存储，所以在跨平台的数据转移中很方便，备份和恢复时可针对某个表进行

InnoDB免费的方案可以是拷贝数据文件、备份binlog或者用mysqldump

④事务

MyISAM强调性能，每次操作都是原子的，不提供事务支持

InnoDB提供事务，外部键等高级数据库功能

⑤AUTO\_INCREMENT

MyISAM自动增长必须是索引，如果是组合索引，自动增长列可以不是第一列

InnoDB自动增长同样需要是索引，如果是组合索引必须是第一列

⑥表锁差异

MyISAM只支持表索引，select update insert delete都是表锁，如果加锁后满足insert并发是可以在表尾部插入，不过要注意的是select加的是共享锁，不排斥其他共享锁

InnoDB支持事务和行级锁，行锁提高了并发能力，但是只有在命中索引的情况下才会进行行锁，否则将锁表

⑦全文索引

MyISAM支持FULLTEXT全文索引

InnoDB不支持全文索引，不过在5.6之后支持

⑧表主键

MyISAM允许没有任何索引的表存在

InnoDB如果没有主键和唯一索引就会自动生成一个6字节的主键，数据是主索引的一部分，附加索引保存的是住宿因的值，注意InnoDB必须由聚集索引，可以没有主键

⑨表的行数

MyISAM保存有表的总行数

InnoDB没有

注意：如果是select \* from table，myisam效率很高，innodb会遍历表，但是如果加了where条件那么myisam和innodb一样

⑩外键

MyISAM不支持

InnoDB支持

1. MySQL的主从复制原理和流程？

主要是应用三个线程完成的：

①主binlog线程：记录所有改变数据库数据的语句放进master的binlog中

②从io线程：使用start slave之后，负责从master上拉取binlog内容放在自己relay log里

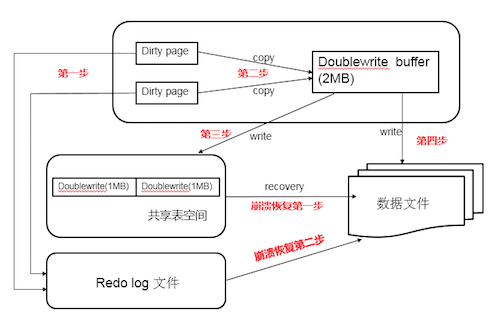
③从sql执行线程：执行relay log中的语句

1. InnoDB引擎的四大特性？

插入缓冲，二次写，自适应哈希索引，预读

①插入缓冲insert buffer：InnoDb使用insert buffer欺骗数据库，对于非聚集索引，并非实时的修改操作并非实时的更新索引的叶子页，而是把若干对同一页的更新缓存起来来做合并一次性更新操作，虽然没有真正把数据方到对应的数据页中，但是仍可以查询到。转化随机IO为顺序IO，提高写的性能。

②二次写Double write：为了解决partial page write的问题，但是这也是分情况的，如果mysql在向数据页写数据的过程中宕机了，那么就发生了partitial page write的问题，但是这种情况一般可以通过redo log 和undo log来完成数据恢复；但是如果这个页以很低概率损坏了的话redolog也无能为力，下面会整理为什么：redolog是针对数据页的操作。此时double wrtie就可以登场了。当mysql将脏数据flush到data file时，先使用memcopy将脏数据复制到内存中double write buffer中，之后通过double write buffer再分两次，每次写入1MB到共享表空间（磁盘，顺序的），然后再次真正的将数据fsync到磁盘，这样就有了双保险，如果发生为什么可以通过double write的备份恢复数据，也可以通过redo log恢复。



③自适应哈希：mysql的heap存储引擎默认的索引是哈希索引，InnoDB存储引擎提出了另一种自适应哈希索引。咦，InnoDB不是B+树索引吗？是的，只不过InnoDB有自己的优化如果发现通过哈希索引能提高查询性能，那么它就会根据B+树缓冲建立自适应哈希，建立很快且不是将整个表都建立哈希索引。

④预读read-ahead：线性预读和随机预读，线性预读可以通过设置当前extent中多少page被读取就把下一个extent的数据读入缓冲；随机预读是当前extent的page被读取的话就把当前extent所有page读入缓冲。随机预读在5.5中被默认关闭了。extent可以认为是64个page组成的block。

1. MySQL中的varchar和char的区别以及varchar(50)中50的含义以及int(20)中的20？

varchar和char的区别：

char是一种固定长度的类型，varchar则是一种可变长度的类型。

char的长度在0-225之间，保存char类型时，右边用空格填充，检索时去掉尾部空格，索引效率高，可以用在一定确定的字段，比如手机号、身份证号等；而varchar，长度在0-65535之间，但是最大有效的长度由定义的长度和字符集确定，整体最大长度是65535个字节。保存varchar类型值时的长度是实际长度+1，这个1是记录实际长度用的，索引时保留尾部的空格，索引效率不高。

总结：varchar空间上合算，char时间上合算。

关于varchar(50)：

最多放50个字符，varchar(50)和varchar(200)存储”hello”所占空间一样，但是后者排序的时候消耗更多的内存。

int(20)的含义：

指显示字符的长度，跟varchar(20)不同，仍占4个字节，如果长度超过了20那么只显示20位，当然如果设置了未满填0的设置，那么未满20时前面填0，默认不填。

和varchar(20)还有一点不同就是int(10)和int(20)存储和计算都一样。

1. MySQL数据库设计表的三范式？

第一范式1NF：数据库表中的字段都是单一属性的，不可再分。这个单一属性由基本类型构成，包括整形、实数、字符串、逻辑型、日期等。

第二范式2NF：数据库表中不存在非关键字段对任意候选关键字段的部分函数依赖、

第三范式3NF：在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖。

说实话有点蒙，到底怎么回事？简单说一下吧：

①第一范式：其实这个范式没太大意义，因为没有人会违背，比如把开始时间和结束时间放在一个字段(2019-05-30,2019-05-31)中这种操作很少会有人做，都是开始时间、结束时间分别由自己的字段(2019-05-30),(2019-05-31)。

②第二范式：比如班级名称和教室是一一对应的，此时应该给班级名称做主键索引（或者单独有一个自增主键ID），这样就是完全依赖了（或者跟主键没关系），但是如果有人把班级名称和授课老师两个字段作为组合键，虽然班级名称+授课老师可以做主键，但是教室就只会依赖主键的一部分（班级名称，因为跟授课教师没关系），所以不行，解决办法就是创建自增主键id

③第三范式：在第二范式的基础上，限制不能传递依赖，这个解释起来比较麻烦，说白了就是主键和非主键1和非主键2不能存在传递依赖，其实可以用一句话解释！！！！！————主键和非主键可以存在父子关系，但是非主键之间不能存在父子关系！解决这种问题的方法很常见也很有效率———分表。

1. MySQL到底应该知道哪些日志类型，他们有什么作用吗？

①事务日志之一redo log日志

事务日志主要用来保障数据库的ACID，尽可能降低宕机造成内存中数据丢失

事务日志会生成两个文件ib\_logfile0和ib\_logfile1，大小默认是5M，用于保存事务日志，在事务执行过程中数据的更改在未提交前都会先写到事务日志中，由于事务日志是连续的磁盘空间，IO性能高，可以保证数据及时写入事务日志。

事务日志包含redo log和undo log，基于InnoDB存储引擎的mysql之所以可以从崩溃中恢复正式依赖于事务日志，当数据库宕机重启后，重启mysql会自行检查事务日志，然后一次处理：

对于事务日志中未正常提交的事务，则会记录到undo log中，因为事务未正确执行完，必须回滚从而保证数据一致性；对于事务中已正常提交但未同步持久化到磁盘上则会记录到redolog中，mysql会重新执行一遍事务。

注意：：：redolog和undolog都是针对数据页的操作，比如redolog记录page number of xxx记录数据”this is abc”，这说明什么，说明了当数据页坏掉的时候redo log也没有办法了！！几率很小，这也是为什么需要double write的原因。

②事务日志之undo log：上面说了

③二进制日志binlog

是一个二进制文件，主要用于记录可能引起数据库数据更改的sql语句，会记录insert update delete grant等语句以及其执行时间、执行时长等额外信息，但是不会记录select show等语句，此时有一个面试题是关于log\_bin和sql\_log\_bin的问题？

log\_bin是只读的全局的配置项，而sql\_log\_bin是会话设置项，sql\_log\_bin只针对当前会话，具体有什么作用呢，比如在恢复数据的时候我们不想将恢复的大量操作记录到binlog中，为了提高性能而已，此时就可以应用sql\_log\_bin，设置为off，这样当前会话就不会记录binlog了。

④查询日志

会记录所有查询日志，不建议开启

⑤慢查询日志

跟查询日志相似只不过会记录查询时间超过long\_query\_time的那些查询语句。不建议开启，当然在调优的时候可以适当开启一段时间。

⑥中继日志relay log

中继日志主要用于主从复制中，在slace节点上开启用于从master同步二进制日志数据

⑦错误日志error log

记录mysql守护进程启动和关闭过程中产生的错误信息，运行中产生的错误信息

注意：mysql的日志都是采用的顺序IO的方式写，这样大大降低了因为写日志导致的性能下降。但是数据和索引信息是随机IO（insertbuffer可能缓解一下随机IO）。

1. 请解释下mysql中的float、double、decimal的区别？

其实从基本数据类型中可以知道float是单精度浮点数，double是双精度浮点数，单精度和双精度的区别在于有效位数，单8双16位。浮点的意思是不精准，存在四舍五入。

回到数据库，当我们建表定义字段类型的时候，他们三个都可以指定精度和标度，精度就是有效位数、标度就是小数点后面几位比如

CREATE TABLE test(f FLOAT(5,2) DEFAULT NULL,d DOUBLE(5,2) DEFAULT NULL,de DECIMAL(5,2) DEFAULT NULL);

INSERT INTO test(f,d,de) VALUES(1.234,1.234,1.234);没问题

结果是

如果不指定标度和精度：

INSERT INTO test(f,d,de) VALUES(1.234,1.234,1.234);

结果是

float和double没问题，但是decimal默认是有精度和标度的（10,0）所以默认decimal没有小数点。

注意：sum()的时候float和double由于是浮点数不精准导致计算结果有很多位小数。

1. CHAR\_LENGTH、LENGTH、BIT\_LENGTH是什么意思？

BIT\_LENGTH：返回字符串的位数

LENGTH：返回字符串的字节数

CHAR\_LENGTH：返回字符串的字符数

可想而知Latin字符串的字符数和字节数是一致的，因为Latin一个字符就是一个字节编码，而对于Unicode和其他编码他们就是不同的。

1. Mysql中的ENUM的用法是什么？

指的是字符串的枚举类型，可以认为是预定义的取值范围

比如create table tt(name ENUM(‘Sam’,’Loe’,’Mike’) default ‘Leo’);如果插入了不是这三个值中的值，那么记录时null值。

1. LIKE和REGEXP操作有什么区别？

REGEXP：正则匹配，子串匹配（不区分大小写，如果想区分需要加上BINARY）

LIKE：完全匹配

1. Mysql的group by的实现原理，如果可以再说说distinct？

group by实际上会进行排序操作，与order by相比，group by主要只是多了排序之后的分组操作。在Mysql中group by有三种实现方式，两种利用了索引，还有一种完全无法使用索引的方式。

①使用松散索引扫描（Loose）

松散索引扫描就是当MySQL完全利用索引扫描来实现group by的时候，并不需要扫描所有满足条件的索引键即可完成操作得出结果。

对于group by引用的多个字段，需满足于所建立索引的最左前缀索引，否则进行group by操作时，无法利用索引。在利用索引时，group by可根据索引，即可对数据分组，此时完全不用去访问表的数据值（索引健对应的数据）。这种实现方式就是利用松散索引。

②使用紧凑索引扫描（Tight）

当group by引用的字段无法构成所建索引的最左前缀索引时，也就是说group by不能利用索引时。如何where语句（如果有的话）弥补了这种差距，比如：group by引用的字段为（c2，c3），而索引为（c1，c2，c3）。此时如果where语句限定了c1=a（某一个值），那么此时mysql的执行过程为先根据where语句进行一次选择，对选出来的结果集，可以利用索引。这种方式，从整体上来说，group by并没有利用索引，但是从过程来说，在选出的结果中利用了索引，这种方式就是紧凑索引。这种方式，mysql的执行计划为using where,use index。而松散索引的执行计划为using index for group by。

③临时文件

如果mysql如论如何都不能利用索引时，此时mysql将读取所有的数据建立临时文件，对文件进行排序，完成分组操作。

关于distinct的实现原理实际上和group by非常类似，只不过实在group by之后的每组中只去除一条记录而已，同样有三种实现方式，仅仅使用索引的松散索引方式、紧凑索引方式以及临时表的方式，一点区别就是group by的临时表方式要进程filesort，不然没法分组，而distinct不用filesort，因为它只需要一条。

1. mysql缓存是如何实现LRU缓存淘汰策略的？

我之前只知道redis缓存的LRU，其实现在想想，Mysql同样也有缓存啊。

所以在介绍redis实现缓存的LRU之前先说一说mysql的缓存吧。

mysql的缓存机制：

说白了就是缓存sql文本及结果集。用kv形式保存在服务器内存中，如果运行完全相同sql则会利用缓存，不会再去解析、优化、执行了。如果这个表修改了那么跟这个表相关的所有缓存都不在生效，显然如果频繁更新的表查缓存不合适而且还会给服务器造成压力。

命中条件：

绝对相同！！！刚才说了缓存在KV的hash表中，通过查询sql、查询数据库、客户端协议等作为key，在判断命中之前不进行解析。对于sql语句，任何的不同包括空格、注释等都会导致缓存不命中！同时如果存在不确定数据如now() current\_date()等这些查询不会被缓存。

工作流程：

①服务器接收到sql，以sql和一些其他条件作为key查询缓存表

②如果找到缓存直接返回缓存

③如果没找到，则进行sql解析、优化、查询等

④执行完sql后，将结果集缓存到缓存表中

缓存失效：

只要涉及表的修改的操作都会让跟表相关的缓存失效，另外就是LRU了等会介绍

缓存的内存管理：

缓存会在内存中开辟一块大小为query\_cache\_size的内存维护缓存，其中有40k空间维护元数据：空间内存、数据表和查询结果映射、sql和查询结果映射，mysql将这个大内存分为小内存块（query\_cache\_min\_res\_unit），每个小块中存储自身的类型、大小和查询的结果集、还有前后没存块的指针。

缓存机制与MyISAM和InnoDB:

需要注意的一点是：MyISAM只缓存索引不缓存结果集，但是os系统级别的缓存就不算了，但是InnoDB索引和结果集都缓存！但是只要遇到加锁的事务，那么该事务中的查询将不会查缓存而是进行sql解析、优化、查询。同时也要注意InnoDB是行锁+MVCC来进行多版本并发的，当前事务ID一定要大于的事务才可以查缓存！其实跟之前说表改变，那么涉及该表的缓存全部失效是一样的。

那么有人会问什么事MVCC？这个问题等会整理。现在来说一下Mysql缓存的LRU吧：

再说Mysql的LRU缓存实现之前先简单说一下缓存过期策略的三种策略：

①FIFO：先进先出，非常好理解，用一个FIFO队列就可以实现，新加入的进入队列尾部，淘汰队首的数据

②LRU：least recently used最近最少使用，用链表实现：新数据插入俩表头部，每当命中缓存就将该数据移动到链表头部，当链表满的时候删除链表尾部数据

如果用hash表+双向链表来实现LRU将更加高效，比如linkedHashmap，查询是否命中显然hash快，插入删除数据显然链表更快，所以...

但是LRU有一个缺陷就是当出现批量插入时容易造成缓存污染，应该可以理解吧，所以更优的方案是LRU-K，其中K一般是2即LRU-2，意思就是访问2次再移动到链表头部

③LFU：least frequently used最近频繁使用：用队列+计数实现，新数据加入队列尾部，计数为1，当队列中数据被命中时引用计数+1，同时队列重新排序，当需要淘汰数据的时候把队列尾部数据删除

知道了淘汰算法之后，实际上不管是redis还是mysql实现缓存淘汰的方式绝没有这么简单，所以还是有差异的，那么mysql到底是怎么进行LRU淘汰缓存的呢？

mysql用一个链表来实现LRU，跟上面介绍的差不多，但是区别在于每次新缓存进来的时候并不是插到链表表头而是5/8处，为什么？因为这个链表被分为young区和old区，old区占整个链表的3/8，每次插入到old区的开始，这样就会把old区尾部挤掉，如果old区的缓存被命中，就把这个数据移动到young区的表头，这样就让young区很少使用的向old区移动了。

顺便说一下redis的LRU缓存淘汰实现吧：

redis既支持LRU也支持LFU，对于LRU的实现redis是近似的LRU目的是节省内存。

可以通过设置maxmemory来设置开启缓存淘汰，如果设置为0，表示不限制内存，那么就不会触发缓存淘汰。近似的LRU实现就是按照预先设定好的一些参数，比如哪些key参数淘汰，比如选择有过期时间的key参与淘汰、所有key参与淘汰等多种方案，同时还会根据一个sample参数进行淘汰，这个sample参数越大越接近真实的LRU，所以不要太大，默认是5。

1. 什么是InnoDB的MVCC？

MVCC(Multiversion Concurrency Control)，即多版本并发控制技术，它是的大部分支持行锁的事务引擎不在单独的使用行锁来进行数据库的并发控制，取而代之的是把数据库的行锁与行的多版本结合起来，只需要很小的开销就可以实现非锁定读（非阻塞读），从而大大提高数据库性能。

已经知道的是，加了读锁后不能再加写锁，加了写锁后不能加任何锁，表锁开销低并发弱，读的表锁可以再读而写的表锁读写都不行，行锁开销大，并发强存在死锁。

MVCC主要是为Repeatable-Read事务隔离级别做的。主要实现原理：

InnoDB存储的最基本单元row中包含了一些额外的存储信息：

①DATA\_TRX\_ID，6个字节标记了最新更新这条记录的transaction id，每个事务自动+1

②DATA\_ROLL\_PTR，7个字节指向当前记录项rollback segment的undo log记录，找之前版本的数据就是通过这个指针

③DB\_ROW\_ID，6个字节当由InnoDB自动产生聚集索引时，聚集索引包括这个值，否则聚集索引中不包括这个值，这个用于索引中

④DELETE BIT位用于表示该记录是否被删除，这里的不是真正的删除数据，而是标记出来的删除，真正意义的删除实在commit的时候

具体执行过程：

begin -> 用排它锁锁定该行 -> 记录redo log -> 记录undo log -> 修改当前行的值，写事务编号，回滚指针指向undo log中的修改当前行

而InnoDB实现MVCC主要依靠两个附加字段：创建（修改）版本号、删除版本号

①insert

InnoDB为每个新增行记录当前系统版本号作为ID

②delete

InnoDB为每个删除行的巨鹿当前系统版本号作为行的删除ID

③update

InnoDB复制一行，这个新行的版本号使用了系统版本号，同时把系统版本号作为删除行的版本，说白了就是insert+delete

④select（关键）

InnoDB会检查每行数据，确保符合两个条件：第一：InnoDB只查找版本早于当前事务版本号的数据行，确保读已提交；第二：行的删除操作的版本一定是未定义的或者大于当前事务版本号的，确保行没有被删除

这样就实现了非阻塞读操作。

注意：InnoDB的MVCC并不是真真正正的MVCC，因为InnoDB在进行行操作的时候是有排它锁的禁止其他事务操作该行！然而理想的MVCC是很难实现的，因为要保证ACID。

1. 聚集索引与非聚集索引？

聚集索引（聚簇索引）clustered index：数据的行的物理存储顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表只能有一个聚集索引。如果表中没有明确创建聚集索引，那么innodb会自动创建一个聚集索引。

聚集索引的叶子结点会存放记录的整行数据，所以使用聚集索引查询数据肯定是只查询一次！

非聚集索引unclustered index：与聚集索引相反，存储顺序与逻辑顺序不一致，一个表中可以有多个非聚集索引，

非聚集索引的也直接点存放的是聚集索引的值，目的是如果非聚集索引的select没有索引覆盖的话就利用聚集索引的值进行二次查询（注意二次查询的速度要慢50%，所以有时候mysql会进行优化如果发现进行全表扫描更快就不会用索引）。

那么什么是索引覆盖？

所以覆盖就是select的列里面包含条件中的索引列或聚集索引列或联合索引列。

举个栗子：

id是主键、name是索引、(age, hobits)是联合索引

select id,name from table where name=’aaa’;

select name from table where name=’aaa’;

select age,hobits from table where age=’25’;

## redis部分

1. 什么是redis以及有什么主要的特点，有什么应用场景？

Redis(Remote Dictionary Server)是C语言开发，基于内存的key-value数据库。

①速度快——内存、C语言、单线程

②丰富的value存储结构：string、hash、list、set、zset、bitsmap

③客户端语言实现多

④持久化：AOF RDB mix

⑤主从复制

⑥高可用——sentinel哨兵

⑦分布式——cluster

应用：

①缓存（会话缓存、数据缓存）

②Pub/Sub

③分布式锁

④分布式队列

⑤排行榜

既然说了redis丰富的value存储结构，那么就说一下他们类型、特点、应用场景吧？

①string

set、get、decr、incr、mget

用于计数：微博数、粉丝数

②hash

hget、hset、hgetall

其实就是双层结构被，就是再次封装，所以你认为是包含于被包含的关系就可以用hash

③list

lpush、rpush、lpop、rpop、lrange

双向链表，消息队列、存储微博关注列表、粉丝列表等

④set

sadd、spop、smembers、sunion

实现用hashmap实现的，跟hash数据类型一样，其实跟list差不多，先进先出但是区别有两点，一个是去重，第二个是重置，比如先插入顺序 a b b a ，正常来说只保存a b 但是由于插入顺序，运行spop时出来的顺序是b a

⑤sroted set

zadd、zrange、zrem、zcard

相较set而言，多了一个根据score排序，在实时场景如直播中的排行，比如粉丝数排行、礼物排行

⑥bitsmap

位图，肯定不陌生，在处理海量数据不重复位图、判重的bloomfilter已经认识到了它的强大，redis用bitsmap最主要的应用就是海量不同用户的计数！比如今天登录过的用户，瞬间就可以计算出来

⑦stream

5.0引入的，等会专门介绍一下吧，尤其是跟kafka、rabbitmq的对比

1. redis持久化机制请详细说一下？

①快照snapshots

默认情况下使用快照方式进行持久化，写入二进制文件dump.rdb，可以自行配置持久化策略，多少秒进行多少操作及进行快照，或者手动调用save bgsave

原理：redis进行forks，子进程将数据写入新的rdb文件，替换老rdb文件，从实现上讲：子进程dump数据的时候采用copy on write的方式，如果此时有新的client操作请求，并不会改变需要dump的数据而是改变一个copy备份数据。

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

RDB方式虽然可能出现间歇期数据丢失问题，但是大量数据重载效率很高。

注意：手动调用save的方式并不是上述forks子进程的方式而是在主线程中操作，所以会阻塞服务！

②AOF

以同步日志的方式记录操作，以类似WAL的方式将每次操作日志（写、删除、更改）以追加的方式写入.aof文件。目的是解决RDB方式间歇期的数据丢失问题。

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no #写日志什么时候sync到文件中没法保证。

每秒同步虽然可能损失一秒数据但是效率是最高的

注意：为了方式.aof文件过大，redis有rewrite的功能，实现起来跟RDB的方式非常类似，比如redis收到rewrite的时候，redis forks子进程，根据内存中的数据，往临时的aof文件中写入重建数据库状态的操作命令，此时父进程并不影响client的请求，但是需要将rewrite期间发生的写删命令缓存起来待rewrite操作完成追加到后面，保证了数据的一致性，最后用临时文件替换老aof文件

③mix——两者结合

4.0版本引入，区别在于aof每次rewrite的时候并不是像上述类似RDB的方式，而是将.rdb文件的内存以操作命令的方式写到aof的临时文件中，然后继续向这个aof文件追加dump之后的操作命令，这样就缓解了aof恢复慢，rdb间歇期丢失数据的问题。

1. redis集群模式请详细介绍一下，顺便说一下主从模式那就更好了？

首先说一下为什么要引入或者说为什么要使用集群模式，难道单机模式或者读写分离不行？

说集群模式之前先介绍一下redis的主从模式：

需要注意的一点是，redis单一的主从模式（比如读写分离）他的目的是缓解master的压力，将读的请求分散到所有slaves上。

主从模式：星型模型、线性模型

星型模型：读写分离，数据同步高效，但是主节点切换需要用哨兵

线性模型：读写分离，数据需要一个一个同步低效，但是主节点变更只需要将下一个节点变成master即可

而集群模式与单一主从模式最大的区别就是，一个cluster可以有多个master-slave，每个master存储的数据是不同的，slave是master的备份，可以读但是不能写。

复制原理——全量复制：

不管是单一主从还是集群模式都会涉及从对主的数据同步，下面说一下复制原理：

当slave启动时候会向master发送sync命令，master收到sync命令就会主动进行rdb的dump，并且缓存dump期间的请求，然后全部发送给slave，然后master每接收一条新的写请求都会同步给slave。

复制原理——增量复制：

看这个名字就明白了，master维护一个历史缓存，slave保存一个offset，这样就可以通过增量复制offset之后的数据即可。

星型模型中的哨兵sentinel：

sentinel的作用：

①监控master和slave是否正常运行

②master出现故障时自动将slave转换为master

redis-sentinel sentinel.conf

使用集群模式的优点：

①多master时由于存储数据不同，节点失效时只会失效一部分数据

②集群模式容灾性更好。一个实例一方面由于使用内存过大可能面临灾难数据失效，另一方面可能面临持久化时间长，阻塞对外服务（单线程）

③一台服务器上开启多个实例，可以充分利用多核cpu和内存

缺点：

使用集群模式最好要配以slave，因为集群模式如果发现某个master不可用而又没有salve来替代，那么整个集群就失效了！

1. 为什么单线程的redis如此之快？那为什么还要有nginx多进程的方式或者memorycache多线程的方式？

①完全基于内存，操作时间复杂度是O（1）

②单线程避免了cpu上下文切换和竞争消耗，不用考虑锁的问题，不可能出现死锁

③使用多路IO复用模型

mysql、meory cache、oracle（windows版）——单进程多线程

oracle（linux版）、nginx——多进程

nginx不涉及进程间、线程间数据的同步，因为nginx处理的都是不相关的数据，所以用多进程

1. redis缓存淘汰策略？

在整理mysql缓存淘汰、linux缓存淘汰的时候已经介绍过redis的淘汰机制了，linux采用双链表的方式进行LRU淘汰策略，mysql采用单链表的方式，但是插入是从5/8处插入。而redis当时整理的是近似LRU的淘汰策略。

六中缓存淘汰策略：

①volatile-lru：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰

②volatile-ttl：从已设置过去时间的数据集中挑选将要过期的数据淘汰

③volatile-random：从已设置过期时间的数据集中任意选择数据淘汰

④allkeys-lru：从数据集中挑选最近最少使用的数据淘汰

⑤allkeys-random：从数据集中任意选择数据淘汰

⑥no-encition（驱逐）：机制驱逐数据——永不回收

在mysql面试题中整理redis缓存淘汰时说了LRU是近似的LRU，什么意思？

意思就是当sample值越大约接近真实的LRU算法，比较小的时候LRU算法就接近ttl的淘汰策略，也就是按照过期时间排序，删除快过期的数据。因为真实的LRU是需要内存来保存链表等数据结构的，所以为了节省内存以及数据移动的性能消耗。

1. redis的hash槽你了解多少？

redis集群的键空间被分割为16384个hash槽slot，集群的最大节点数量也是16384个，关系是cluster > node > slot > key，redis cluster在设计上没有使用一致性hash算法，而是使用数据分片引入哈希槽来实现。存储cluster中的所有key都会被映射到这些solt中，集群中每个键都属于这16384个哈希槽中的一个，slot=CRC16(key)/16384来计算key属于哪个槽，按照槽来分片，通过每个节点指派不同数量的槽可以控制不同节点负责的数据量和请求数。

说白了，集群中有多少个master就会将16384平分给这些master（默认）。

既然这样的话不禁会问：集群的水平扩展和压缩，说白了就是数据迁移是怎么做的？

实际上不管是添加机器还是删除机器都是slot槽位和key的迁移，在redis cluster中我们只需要将solt迁移过指定node即可，key自动迁移！

cluster的所有节点都会保存两个数据：

①16384位的二进制序列

②16384个元素的共享数组

第一：其中共享数组是所有nodes都相同的包括master 和slave，它保存的是一个映射关系，告诉每一个node，每个槽位存在哪个master node上，这样集群所有的node都可以接受读写请求并可以相互转发。第二：二进制序列在每个master node上都是不同的，在每个slave node上都是相同的000000...序列，因为slave不进行写，这个二进制序列类似位图，值为1对应的索引数就是本master node存储的槽位值。所以所有master node这个二进制序列不能相同。

空槽位数据迁移：（假如要把192.168.5.111:8000节点中的1000槽道迁移到192.1.68.5.111:8001节点中）

①8001 > cluster setslot 1000 importing 8000（源节点id，cluster nodes 命令中会得到对应的节点id，为了方便用端口号代替）

在目标节点8001上运行从8000节点导入1000号槽位，此时8001中共享数组对应的1000号槽位从正常状态变成importing

②8000 > cluster setslot 1000 migrating 8001（目标节点id）

在源节点8000上运行从8000上的1000号槽位迁移到8001，此时8000中的共享数组对应的1000号槽位从正常状态变为migrating

③8000 > cluster setslot 1000 node 8001（目标节点id）;8001 > cluster setslot 1000 node 8001（目标节点id）;8002 > cluster setslot 1000 node 8001（目标节点id）

所有节点上运行通知槽位的变更，这样所有节点的共享数组变化，8000和8001节点上的二进制序列也会发生改变。

带有数据的槽位数据迁移：（如上的目的）

比如有一个key : set a abc，这个a存储在slot[110]号槽位上

基本步骤如上，只不过在通知所有节点槽位变更之前需要将key也迁移过去，具体实现就不说了

1. redis的事务你了解吗，它和watch如何实现乐观锁CAS，跟mysql中的事务有什么区别？

①MULTI

用于标记事务块的开始，redis会将后续命令逐个放入队列，然后才能使用EXEC命令原子化执行这个命令序列，执行MULTI返回ok

②EXEC

在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态，在这期间不会执行其他client的请求

注意：当使用WTACH命令时，只要当受监控的键没有被修改时，EXEC命令才会执行事务，这种方式也就是CAS（check and set），执行EXEC返回一个数组，其中的元素分别是原子事务中每个命令的返回值。当时用WATCH命令时，如果事务执行终止，那么EXEC命令返回null

③DISCARD

清除所有先前在一个事务中放入队列的命令，然后恢复正常的连接状态，如果使用了WATCH命令，那么DISCARD命令就会将当前连接监控的所有键取消监控，DSICARD命令返回ok

④WATCH

当某个事务需要按条件执行时，就要使用这个命令将给定的键设置为受监控的，返回ok

⑤UNWATCH

清除所有先前为一个事务监控的键，如果调用了EXEC或者DISCARD命令就不需要手动调用UNWATCH，返回ok

其实对于事务我们关注的是原子性——全部成功或者回滚的问题。redis中是不是像mysql中事务的那样有acid的性质呢？是否是原子的；是否有一致性（说白了就是符不符合计算逻辑）；是否是隔离的；是否是持久化的。想一想其实关键的就前两个：到底是不是要么全部成功要么全部回滚，因为redis的单线程保证了隔离性，aof和rdb保证了持久性。

实际上redis不支持回滚！也就是说执行过程中又失败的命令不会影响其他命令！

第一种情况：MULTI开启事务后到EXEC执行事务之间的命令如果出现明显的语法错误时redis执行返回error然后丢弃事务

第二种情况：MULTI开启事务后到EXEC执行事务之间的命令如果没有语法错误，但是中间有命令执行失败了，那么redis会执行其他执行成功的命令！所以redis保证原子性——全部执行但是没有保证一致性！不支持回滚。

关于WATCH命令刚才说过了，是对一个或者多个key进行监控，在调用EXEC之前如果发生改变，那么本次事务将会被丢弃。

Redis通过WATCH的CAS实现乐观锁：

WATCH myKey

MULTI

....

EXEC

如果在调用EXEC之前myKey被其他客户端改变了的话，那么本次失误将不会执行。

注意：redis从2.6开始支持脚本执行，而脚本中的命令是原子的！



这个代码就是利用lua脚本，调用jedis.eval执行脚本并内部和外部参数，这个命令用来分布式锁的解锁过程，因为是解锁，所以要原子的！

1. redis如何实现的分布式锁，还有什么可以实现分布式锁，他们有什么不同之处？

什么是分布式锁，一般怎么实现？

分布式锁指的是在多个客户端同时请求对数据的占用时加锁，那么为什么是分布式，因为当单个进程多个线程访问时只需要在程序中加锁即可；一台服务器多个进程访问时可以用进程间通信的方式达到锁的目的，但是多台服务器多个进程同时申请同一个资源时怎么办，在代码层面不好实现锁，所以就出现了分布式锁，将申请释放锁与程序解耦。

分布式锁要保证：

①互斥性，同一时刻只能有一个客户端获取锁

②可重入性，当断开连接后可以继续获取锁

③不能死锁

④容错性

实现的方式一般有三个：

①基于数据库实现分布式锁

②基于redis实现分布式锁

③基于zookeeper实现分布式锁

下面开始一一介绍：

①基于数据库

其实用数据库的方式具体将应该是两种方案：

第一种：基于数据库表的增删等操作

当需要获取某个方法的锁时用insert插入一条method\_name是唯一索引的一条记录，当然可以携带当前服务器、进程号、线程名等记录实现可重入，此时利用天然的唯一索引的功能实现了只有一个线程能获取锁，当使用完毕之后进行delete操作。但是这种方法一个很大的问题就是不能阻塞线程、释放锁之后无法唤醒其他等待线程。当然可以通过一个死循环一直尝试insert达到非阻塞的目的，但是太麻烦了，有一个非常严重的问题就是服务器宕机，插入的数据没有删除，也就是这个锁其他人永远获取不到了。

第二种：基于数据库行记录的排它锁

select .......where method\_name=’...’ for update;这样可以锁定这条记录，其他线程无法对其

修改，当需要释放锁的时候commit即可，该方式可以实现阻塞，因为不像insert，插入失败就返回错误，而for update会阻塞一直等待锁！比使用数据库表的方式高明一些，可以自动释放锁，但是有一个极大的问题就是mysql到底用不用索引来锁定某些行，还是直接锁表是mysql说了算，如果直接锁表了，那就悲剧了。

②基于redis缓存实现

jedis.set(String key, String value, String nxxx, String expx, int time)

key是用来当锁，因为在redis中国key是唯一的，可以代表共享的唯一资源，其他人获取锁

value可以传入requestid，为了实现可重入性

nxxx：NX意思是set if not exist，当key不存在是进行set操作，如果存在不进行操作

expx：PX意思是要给这个key加一个过期时间，具体时间由下一个参数决定，目的是方式死锁，可能由于断开连接后永不释放这个key锁

time：过期时间

关于释放锁的方式，强烈建立使用lua脚本+jedis.eval(脚本)的方式保证原子性！redis的每一个操作虽然是原子的，但是多个操作就不能保证原子了。

③基于zookeeper实现

利用临时有序节点来实现分布式锁

在指定目录下创建临时有序节点（可以带上连接信息保证可重入性），检查自己是不是最小的id，如果是就获取锁，同时其他节点对父节点进行监听，发现节点变更就查看自己是不是最小的那个。

1. 请说一下什么是缓存穿透、缓存击穿、缓存雪崩？应该怎么解决或者避免？

按容易理解程度一个一个来：

①缓存雪崩

雪崩指的是在某一时刻大量的key失效，导致大量的访问去请求mysql造成压力，可能是因为redis节点的失效、slave并没有升级为master导致整个redis集群宕机或者是因为过期时间大量相似。

解决方案是：key过期时间+随机时间/一致性hash来保证集群的稳定，但是需要复杂的架构

②缓存穿透

指一直查询一个不存在的值，不存在的话缓存中必然没有，所以每次都会去mysql查！

解决方案：如果mysql没有就在redis中缓存一个=null的key/如果确定某些key不可能存在就在业务层进行拦截

③缓存击穿

指的是某个或者某些少量的key超时过期了，而此时大量对该key的请求全部落在mysql持久层造成数据库卡死。

比如典型的秒杀活动，如果设置不当在秒杀开始前缓存失效的话....

解决方案：对某些热点key设置长的超时甚至forever/人为干预一些热点key...

三者的区别：雪崩——大量key失效，击穿——少量key失效，穿透——一直请求没有的key

下面还会介绍一下缓存并发的问题！

1. 刚才你提到redis单线程不存在死锁的情况，也就是不存在key的竞争问题，但是可能存在key的并发问题！所谓并发就是可能存在多个set操作虽然都能执行，但是可能存在执行顺序的问题，比如正常 1 2 3 4 5 这个顺序set key，加入顺序变了 1 3 4 5 2，此时结果就不对了，redis怎么解决这种情况？

解决方案：

①分布式锁+时间戳

分布式锁的作用是只有一个client在操作，然而redis的key并发set操作还需要一个保证就是set的顺序，分布式锁不能百分百保证顺序，所以需要一个时间戳放进value，如果发现当前操作的value中的时间戳小于已经set完毕的value中的时间戳，那么本次set操作将pass

②消息队列

使用消息中间件用来串行化操作

1. redis中5种数据类型底层是什么数据结构实现的？

redis中5中数据类型不用多说了：string、list、hash、set、zset

redis底层有6中数据结构：

①SDS动态字符串

②链表

③字典

④跳跃表

⑤整数集合

⑥压缩列表

下面一一介绍一下这6中底层数据结构的特点，以及他们用来是什么哪种redis数据类型：

①字典hashTable

为什么首先说hashTable因为hashTable是redis中最常见的数据结构，因为redis底层key和value的存储全部都是用hashTable存储的！hashTable再底层的实现就是数组+链表。

另外hashTable用来作为redis的hash数据类型，可以扩容和缩容（渐进式rehash），扩容之后并不会直接进行数据迁移，而是待后续进行新的写、更改、删除操作的时候在新的数组上操作，然后从原来的数组中删除原来的数据，当然redis也可以定时进行数据迁移。

②SDS动态字符串

SDS数据结构有char[]用来保存字符串、free保存未使用长度、len保存已使用长度，特点是空间预分配和惰性空间释放。如果小于1M的字符串会分配和len属性相同大小的未使用空间，如果大于1M会给字符串多分配1M的未使用空间，防止频繁的内存申请和释放，其次惰性空间释放就是说当字符串长度缩小的时候不会立即回收未使用空间而是记录在free中。

③链表

用来作为list，FIFO

④跳跃表

有序的数据结构，大部分情况下可以喝红黑树相媲美，而且实现起来比红黑树简单，而且关键是跳跃表可以实现区域读取，而红黑树不行

⑤整数集合intset

当一个集合只包含整数元素，并且这个集合的元素数量不多的时候会先使用inset，如果出现不全是整数并且数量比较大的时候就会将集合升级

⑥压缩列表

每个节点保存了前面节点的长度和最后节点的位置，有效的向后遍历。



可以说：redis中5中数据类型每种都有至少两种底层实现方式，目的是内存、性能。

在说5中数据类型的实现之前先看一下redis中任何一个value（实际上就是一个redisObject）的对象构成：

type：value的数据类型，0代表string，1掉膘list，2代表set，3代表zset。4代表hash

encoding：底层实现的数据结构，比如0代表encoding\_raw，1代表encoding\_int，2代表encoding\_zipmap，4代表encoding\_linkedlist，5代表encoding\_ziplist，6代表encoding\_inset，7代表encoding\_skiplist，8代表encoding\_embstr

lru：如果开启了masmemory设置，那么redis就会有lru记录最近一次访问时间

refcount：用于引用计数和对象共享。引用计数可以理解，刚被创建时是1，如果变成了0说明没人用它了可以被回收，另一个作用是对象共享：在redis初始化服务器的时候会创建0-9999的字符串对象用于对象共享，当使用set命令创建一个新字符串对象时，如果要创建的字符串已经存在则将指向的指针指向该字符串对象。

①String类型的实现

第一种实现：INT编码：当字符串保存的是一个可以用long类型表示的整数时用它

第二种实现：EMBSTR编码：最目前最新版本，当字符串保存的是一个小于等于44个字节是用，说白了就是短小字符串用它，如果后续要改动这个value，那么会自动转为RAW编码

第三种实现：RAW编码：字符串大于44字节的时候使用。

②List列表

第一种实现：Ziplist压缩列表：当列表对象保存的字符串元素都小于64字节，并且元素数量小于512个时

第二种实现：Linkedlist双向链表：除了上面说的就用linkedlist

③Set集合

第一种实现：intset整数集合编码：集合对象保存的所有元素都是整数型并且数量不超过512个

第二种实现：hashtable编码：除了上面的情况用hash表，value全是null

④Zset有序集合

第一种实现：ziplist压缩列表：元素数量小于128个，并且元素成员长度小于64字节，使用ziplist，关键点在于每个元素使用两个压缩列表节点保存，第一个保存成员，第二个保存分数，集合元素按分值从小到大排序、

第二种实现：skiplist跳跃链表：注意并没有那么简单，而是skiplist+hash来实现的zset，skiplist保存分数的递增序列，无论查询还是增删对于跳跃链表都是logn的，而hash表用来保存分数和value的映射的。是O（1）的

⑤hash表

第一种实现：ziplist压缩列表：哈希对象中保存的所有键值对的键和值的字符串长度都小于64字节，并且键值对数量小于512个，使用ziplist

第二种实现：hashtable，不多说

1. Jedis与Redission的区别，官方为什么会建议使用Redission？

①概况对比

jedis是java实现的客户端，其中api提供比较全面的redis命令支持，Redission实现了分布式和可扩展的java数据结构，功能比较简单，宗旨是对redis的分离，将精力集中的业务逻辑上。

②编程模型

jedis基本与redisapi保持一致，redission比较抽象，一个方法可能会调用几个redis命令

③可伸缩性

jedis是同步阻塞IO，调用都是同步的，不支持异步，jedis不是线程安全的，所以需要通过连接池来使用jedis；redission使用非阻塞IO和基于netty框架的事件驱动模型，方法调用是异步的。redission的api是线程安全的，所以可以操作单个redission连接来完成各种操作

④数据结构

jedis只支持string hash list set sortedset，redission不仅提供一系列的分布式java常用对象，基本可以与java的基本数据结构通用，还提供分布式服务。

1. Redis5.0出现的stream你了解吗，听说是流数据消息队列，你怎么看待它的出现以及它和kafka有什么不同？

实际上可以认为stream的出现就是一个消息队列，而之前redis本身有list、PUB/SUB可以作为消息队列。但是他俩都有很大的弊端。

Pub/sub：订阅和推送，这种方式有一个极大的弊端就是当断开连接之后，期间的数据不会再次被消费，Pub/Sub的数据不会被持久化，所以消息容易丢失。

List：这种方式当断开连接之后虽然可以继续消费以及可以持久化，但是一个list只能被消费一次！消费完就删除，如果采用多个list的话那么可拓展性太弱了。

Stream的出现就是要解决上述问题。

stream中保存的消息是按照seq\_id顺序排列的，redis会记录stream里每个消费组里最后消费的last\_id及还没有返回的ack确认的id数据pending\_ids。每个消费组都有一个last\_id，也就是说每个消费组都可以消费同一条数据，但是同一个消费组内的消费者不能消费同一个stream的同一条mesasage！

其实从整体上看跟kafka差不多，但是细节有很多不同之处，stream个人感觉唯一一点好处就是有序，但是kafka如果是单个partition或者高版本的kafka的时间戳机制都可以保证顺序性！

1. 聊了这么多redis，说到底redis用在缓存层比较多，那么缓存就一定会面临一个很严重的问题就是缓存和数据库的双写一致性问题？

首先什么是缓存与数据库的双写一致性？如果不一致会存在什么问题？

一致性则是redis中的数据和mysql中数据应该保持一致性，尤其是改变mysql中数据的过程中应该怎么处理redis中的缓存数据这是个难点，不一致造成的问题必然是请求读到的数据是脏数据！

①读操作

先读缓存，如果有就直接取缓存的，如果没有就去查mysql然后将数据库查出来的数据写到缓存中，最后返回给客户端

②更新请求

如果仅仅是读请求不存在数据不一致的问题，但是更新的时候可能存在数据不一致！

第一种选择：先操作数据库，再删除缓存（之所以是删除：懒加载）

这两种方式无论哪种方式都可能存在缓存和数据库不一致的问题！比如第一种选择中先操作数据库再删除缓存，如果删除缓存失败或者在删除缓存过程中有新的请求都会造成数据不一致！

解决方法：把删除缓存的操作用专门的线程来操作，比如把要删除的key发送到消息队列，然后用单独的线程操作直至key删除成功。（还有可能存在脏数据的时间延时）

第二种选择：先删除缓存，再操作数据库

就像刚才说的两种情况都有可能存在不一致情况，比如当一个线程删除完缓存还没来得及操作数据库，另一个线程查询缓存发现没有就去mysql中查，然后缓存到redis中，前一个线程并不会发现这种情况，所以会造成不一致。

解决方法：仍然操作消息队列的方式，将删除缓存、操作数据库、查询缓存等操作全部串行化！

关于消息队列的解决方案还是得多两嘴：

单机版：

需要更新数据的时候根据数据的唯一标识key将操作路由之后发送到一个jvm内部队列中，读取数据的时候如果发现数据不在缓存，那么就将读取数据+更新缓存操作根据唯一标识发送到对应的jvm队列中！

每个队列（根据key区分）都是一个操作线程，串行操作，如果这个过程中如果发现多个更新缓存的操作，那么只需要执行最后一个更新缓存的操作即可。这种方式适用于单机版的情况。如果在分布式情况下每个jvm内部维护很多队列，他们之间如果不进行串行的话同样也会造成数据不一致的问题。

多机版：

其实大多数情况都是多个服务器，因为作为web或者其他查询服务都会部署多个实例，这样的话应该怎么做？

最简单的方案是不同服务器处理不同的数据更新操作，比如根据请求参数中key在nginx请求转发时就进行路由，将操作同一个key的操作路由到一个服务器上！这种方案是最简单的。

对比两种方案：

①先更新数据库，再删除缓存 cache side pattern

在原子操作失败时面临较大问题，高并发场景比较好

②先删除缓存，在操作数据库

在原子操作失败时没有影响，高并发场景不太好

## hbase部分

1. 简单说一下你对hbase的理解？

Hbase是建立的hdfs文件系统之上的分布式的、面向列存储的数据库，起源于google的BigTable论文，主要用作存储非结构化、结构化数据，只有一个主键索引rowkey，海量数据的近实时查询，包括主键查询和主键范围查询和全表扫描。

rowkey是表内全局字典排序的，列族column family是表schema的一部分，可以有多个，在定义表时最好全部定义好，每个列族包含多个列column，可以动态增加而不会造成大面积磁盘数据移动（随机io），时间戳（版本）可以自己生成也可以系统生成，通过时间戳来保存不同的版本，最新的排在前面，当然为了防止由于版本过多造成存储压力和索引管理，数据版本可以进行回收：一种是通过控制版本数n，另一种是保存最近一段时间。最后cell单元是由rowkey+column+version确定唯一单元，由于hbase中字段数据没有类型，全部都是字节码存储。

1. Hbase的查询方法？

刚才说了hbase只有三种查询：全表扫描scan、范围查询scan、rowkey定位get，所以实际上可以认为是两种：

①get

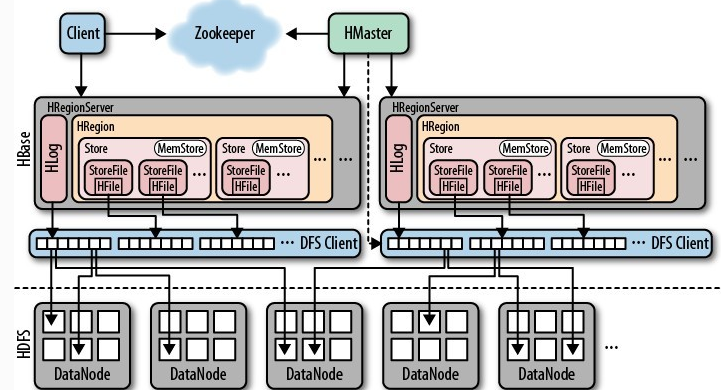
不必多说，根据rowkey来定位查询，当然了可以指定是否应用行的事务性！，说白了就是在查询这个rowkey的时候不允许修改。

②scan

首先是全表扫描，可以通过setCache和setBatch提升查询速度，其次scan可以通过setStartRow和setEndRow来进行范围查询

注意：scan可以通过setFilter方法添加过滤器，这也是分页、多条件查询的基础！

1. 简单说一下Hbase的架构吧？



从架构上看，Hbase包含：Hmaster、HRegionServer、Hregion、Hlog、Store、MemStore、StoreFile、Hfile等，下面就来整理他们之间的关系吧：

先说zookeeper：

第一：肯定是高可用了

第二：储存Hregion的寻址！

第三：监控Regionserver并上报HMaser

第四：存储HBase的表和schema信息

①Hmaster

为HregionServer分配Hregion；负责HregionServer的负载均衡；发现失效的HRegionServer并重新分配；HDFS上的垃圾文件回收；处理table和schema层面的更新请求

②HRegionServer

维护Hmaster分配来的Hregion，处理对这些HRegion的IO请求；负责切分正在运行过程中变得过大的HREgion，实际上这里也说明了client的请求跟HMaser是没有关系的，Client通过zk获取查询数据所在region以及所在regionserver，表是有很多region组成的，这些region由regionserver管理，region上保存的数据格式是rowkey和column family等（下面详细介绍），总之存储的是key value，而value是以列族为单位，而且store对象是存储这些数据来用的，每个store包含memstore和storefile，另外一个regionserver管理多个regions和以这个Hlog

③Hregion

HRegion是存储和负载均衡的最小单元（访问的最小单元），HRegion按照大小来分割，默认256M。关于region会记录对应的表名、startrowkey、创建时间目录表（-ROOT-和.META.）记录endrowkey。

Hregion的存储和读取非常重要，后面会介绍，但是如何定位一个region呢？

HRegion定位：三步走！！：

step1：通过zk里找到-ROOT-表的位置，这个表叫做目录表，这个表只有一个region

step2：根据目录表查找.META.表的第一个表中相应的HRegion的位置，其实.MATA.表的每一个region都是-ROOT-表中的一条记录

step3：通过.META.表找到所要的用户表HRegion的位置，用户表的每个HREgion都是.META.表中的一条记录

④Store

每个HRegion由一个或者多个Store组成，至少是一个Store，HBase会把一起访问的数据放在Stroe里面，即为每个ColumnFamily建一个Store，如果有一个ColumnFamily，也就是有一个Store。一个Store由一个MemStore和0或者多个StoreFile组成！HBase是以Store的大小来判断是否需要切分HRegion！

注意：从这里可以看出什么？数据在内存或者磁盘的存储格式必然是key+value（columnfamily）说白了就是最小存储是列族而不是列。另外容易出错的一点就是一直以为region存储是按照columnfamipy分割的，没想到一个region可以存储多个列族！

⑤MemStore

MemStore是放在内存里的，保存修改的数据及keyvalue，当MemStore的大小达到一个阈值默认（64M）时，MemStore会被flush到文件中，即生成一个快照，目前HBase会有一个线程来负责flush操作

⑥StoreFile

MemStore内存中数据写到文件后就是Storefile，StoreFile底层是Hfile

⑦Hfile

关于HFile还是有点意思的，因为这已经是磁盘文件了，如果才能快速读取想要的数据呢？

HFile：FileInfo+Trailer+DataIndex+DataBlock组成

DataBlock肯定是存储kv数据的了，可以被压缩；FileInfo是记录HFile的元信息的；Trailer保存每一段的偏移量，读取HFile时会先读取Trailer，然后读取DataIndex，这样当检索某个key时不需要扫描整个HFile，只需要从内存中找到key所在的data block即可，通过一次磁盘IO将block读入内存然后获取key value数据。

注意DataIndex采用LRU缓存机制！

⑧Hlog

WAL log：writ ahead log，用来做容灾的，Hlog记录所有有关数据变更的操作，一旦region宕机可以从log中进行恢复，不过有个小问题就是每个regionserver只有一个Hlog，regionserver宕机Hlog也获取不到了啊；另外Hlog不能越来越大吧；还有就是MemStore是遇到阈值就进行写，那么就会有多个HFile小文件存在，如何合并呢？

这几个问题接下来会介绍。

1. HFile文件的合并时机、合并方式等？

其实这个过程是有专有名词的——Compaction。当memstore的数据flush到磁盘后，每次都形成一个Hfile，这个数量很大时就会严重影响HBase读性能，因为文件越多需要缓存的DataINdex越大，并且文件之间是有关联的，比如早一点的文件中一条记录在晚一点的文件中修改了，此时那就需要读出他俩来然后比较version。所以需要合并。

合并作用：

①合并文件

②清除被删除、过期、多余版本的数据

③提高读写数据的效率

合并方式：

①minor：只是用来做部分HFile的合并操作和minVersion=0并且设置ttl的过期版本清理，不做任何删除、多版本合并的清理

②major：是对region下的StorFile下所有Hfile的合并，最终生成一个大文件

合并时机：

这就跟配置和调优有关了

定期检查和每次memStore flush文件时都会进行合并检查，而是否需要进行compact需要用配置中规定的文件个数来决定。

1. Hlog的一些问题，为什么Hlog与regionserver一一对应而不是每个region有一个Hlog？Hlog越来越大怎么办？

首先，HLog肯定跟大多数如mysql binlog、hadoop editslog、redis aof、kafka 持久化都是顺序学磁盘的，所以写的性能可以保证，如果是多个Hlog的话同时写入性能差一些，因为会增加磁盘寻址的消耗。当然了一个Hlog也有弊端就是当regionserver下线的时候需要将Hlog中对应不同的region日志分离出来分别进行恢复。

Hlog是记录数据修改的操作的，如果数据已经持久化了那么Hlog就没有必要记录之前的操作，此时会进行移动为.oldlogs然后删除。

1. HBase的整个读/写流程？

写请求：

①client访问zk，从ROOT表中获取META表中region所在信息，并将其信息写入client cache②client通过三步走会定位到数据所在regionservers，同时向regionserver发出请求

③region检查数据是和schema一致

④如果客户端没有指定版本，则获取当前系统时间作为数据版本

⑤将更新写入Hlog然后写入memstore

⑥判断memstore是否需要flush为storeFile，如果发生了flush的话再判断是否需要compact

⑦如果发现region大小（store大小）超过阈值就会进行split分裂，根据middle key进行分割为相等两份——负载均衡注意：split发生前后都会通知Hmaster

读请求

①同样跟写请求一样从root和meta表中获取需要请求的regionserver

②想regionserver发出读请求

③先从memstore中找，入不到就从Hfile中读取

1. HBase的LSM-Tree模型是什么，干嘛用的？

LSM-Tree是一种索引模型，类似mysql中的hash索引和b+树。

①hash索引：是哈希表的持久化实现，支持增、删、改以及随机读取操作，不支持顺序读取，在数据量小时效率非常高复杂度是O(1)

②B+树：常用的索引结构，它支持增、删、改、顺序查操作复杂度是O(logn)

③LSM-Tree（Log-Structured merge tree）：从名字中都能看出大量顺序写的优势！其实它和B树一样支持增、删、改、顺序查的操作，它的不同支持就是通过批量存储来规避磁盘随机写入的问题，当然也有弊端，LSM树牺牲了部分读的性能。

想了想还是仔细说一说B树、B+树、LSM树吧，不然理解不到为什么说提高了写性能，降低了读性能。

介绍B树之前要先说一下B树之前的二叉树如二叉查找树、二叉平衡树和红黑树，他们都是典型的二叉查找树，时间复杂度是O(logn)，与树的深度有关，所以如果能够降低树的深度可以提高查找效率。

如果对于二叉树来说，在海量数据的情况下，面临一个问题就是每个节点都要存储很多数据，而每个节点又不能存储过多的数据量，不然就会出现每个节点的线性查找问题了。然而如果不存储过多数据情况下，也就是说节点要多才能存储得下海量数据，那么此时树的深度就会变得很高，查找效率必然低下，如果能够采用多叉树结构是不是可以提高查询效率呢？

当然了平衡二叉树出现的目的是为了避免由于数据本身的问题导致节点不平衡，进而导致树的告诉过高，所以最好的就是多叉平衡树——B树！

首先要明确一点，磁盘的读取写入是以磁盘块为单位，所以节点大小最好不要超过磁盘块的大小。

下面说一个问题吧：一颗含有N个总关键字数的m(先认为=3)阶B数的最大高度是多少？

解这个问题之前先看一下N个节点的平衡二叉树的高度：

第一层是1个，第二层是2^1，第三层是2^2....所有假设层数是x，那么1+2^1+2^2...+2^x-1=N，根据等比数列求和公式可以知道2\*(1-2^x-1)/(1-2) + 1 = N,所以2^x = N+1，求对数x=log(N+1)。回到B树，题目已经给出是m阶，所以每个节点的子节点最多是m个，第一层是m-1个关键字，第二层是m^1\*(m-1)，第三层是m^2(m-1)....以此类推，得到(m-1)(1+m^1+m^2...+m^x)=N，所以m^x-1=N/(m-1)，所以答案是x=lomg(N/(m-1)+1)？？？？？答案是错误的！！！这里问的是最高高度不是最小高度

注意刚才二叉树计算高度的时候是按照平衡二叉树计算的，而平衡二叉树是满二叉树，最关键的是二叉树不存在每个节点存储多个关键字的问题只能存储一个关键字！为什么？因为二叉树的每个节点的子节点最多是两个，如果每个节点存储关键字超过一个，那么子节点就必须大于两个，因为这是索引啊！！比如某节点的关键字是2，那么这个节点的左节点小于2，右节点是大于2的树，如果这个节点除了存储2还存储了5，那么你现在告诉我二叉树的两个节点如何区分小于2，大于5,2和5之间的三个分支！！————所以说二叉树的每个节点（叶子和非叶子都只存储一个关键字！！）

然而此时并不是二叉树而是B树：

①树中每个结点至多有m棵子树（即至多含有m-1个关键字）。

②若根结点不是终端结点，则至少有两棵子树。

③除根结点外的所有非叶结点至少有【m/2】(向上取整)棵子树（即至少含有【m/2】-1个关键字）

那么现在就有个问题：节点到底是存储m-1个关键字还是存储1个关键字？如何m等于2个话B树就退化为满二叉树！如果m>=3时，那么就要探讨这个问题了：

①第一种情况：求B树的最大高度

意思就是除叶子节点以外的节点按照最小可能存储关键字数，根据定义，第一层一个根节点最少可以存储1个关键字，那么第二层两个节点，此时除了根节点和叶子节点外，其他节点至少有m/2向下取整个孩子（定义，说实话我也不知为什么），第三层至少有2\*[m/2]向下取整个节点，以此类推，第x层至少有2\*[m/2]^(x-2)个节点，所以第x层的节点数是2\*[m/2]^(x-2)，关于这里说实话我真的是蒙了。直接给答案吧，我是真不知道为什么了lo┌m/2┐g((N+1)/2 )+1。

②第二种情况：求B树的最小高度

意思是包括根节点和非叶子节点，他们保存的关键字全部都是m-1个！也就是说孩子树都是取最高m。所以跟一开始的答案是一样的x=lomg(N/(m-1)+1)，验证一下：如果m=2，x=log(N+1)，答案跟二叉树一致。

现在说完B树了，那么B+树又是什么，为什么B+树比B树更加适合做索引？

B+树与B树的区别在于，节点内只存储指向下一个节点的指针，并不存储关键字（数据）的指针，所有数据的指针全部在叶子节点，这样降低每个节点存储的信息量，从而每个非叶子节点可以存储的索引指针就会越多，每次IO读入内存的索引数据就会约多从而降低IO次数。

从这里看出B+树的读取相当快，写入的时候可能就要涉及数据的移动！这种方式也是为了读的时候更加有效。

然而LSM树是怎么做的索引呢？

为什么他的写效率高？笼统来说就是顺序写，而B+树是随机写。

说白了，LSM树是多个B+小树，内存中有树，磁盘中也有树，而B+树只有一个，无论是加载到内存还是在磁盘中，他就是一个。是分层的，内存中有一个小B+树，磁盘中有多个（多层B+树）注意：发生major compact的时候磁盘文件就会变成一个，此时也会合并B+小树为一个大的B+树，当内存数据超过阈值的时候，将数据和B+小树flush到磁盘中。这样的话可以想象一下，每次写入的时候只需要改变内存中的数据和B+小树，速度很快，不用把磁盘中所有数据和B+树加载到内存中修改树，而flush的时候也不是每次都compact，只要在超过配置文件数量的时候才进行合并操作！那么为什么说牺牲了读取的性能？因为当内存中找不到想要的数据的时候就得加载磁盘中的B+树，所以需要读取很多磁盘文件进行数据合并才知道这个数据的最新版本！！所以磁盘Hfile的合并势在必行！

1. HBase的rowkey设计？

rowkey是HBASE中唯一的索引，也是查找数据的根据，那么rowkey到底应该如何设计呢？

①rowkey长度原则

实际应用中最好控制在10-100bytes，一般来说越短越好，不要超过16bytes，因为64位系统内存8字节对齐，8的整数倍利用了操作系统的最佳特性，并且hbase会将一部分数据载入内存，所以长度越小可以存入内存越多。

②rowkey散列原则

如果rowkey按照时间戳的方式递增，不要讲时间放在rowkey的前面，建议将rowkey的高位字节采用散列字段处理，由程序随机生成。低位段放时间字段，这样讲提高数据分布均衡，哥哥regionserver负载几率相等。如果不这样的话，某时段的数据的读和写都将集中在某个regionserver中，造成热点问题，降低效率——这里主要是查询效率，因为写的话其实还好。

③rowkey唯一原则

必须在设计上保证唯一性，rowkey是按照字典排序存储的，因为涉及rowkey时要充分利用排序的特点，将经常读取的数据存储在一起，将最近可能被访问的数据放在一起。

从以上原则中可以看出，散列一般来说一定要添加，但是一旦是随机生成的，那么就会出现相关数据不可能放在一起，如果想要相关数据存在一起，就要放弃一定的散列原则，所以rowkey的设计上确实很难把握。通过阅读发现，其实rowkey最好是平衡随机性和有序性（有序性就是利用rowkey的字典排序，相关数据在一起的性质），将(唯一标识+时间)的哈希值作为随机数或者像手机号这种后四位的随机性可以设计rowkey。下面介绍几种实现上述原则的方法：

①加盐salting

说白了就是在rowkey的高位添加随机数

②预分区

说白了就是提前根据hbase集群的规模（以及预计以后的规模）来设计出rowkey分布规则，比如现在有10台服务器（后续可能有100台），那么设计加盐的随机数时仍然设计0-199，然后依据现在的10台的规模划分区间0-10、10-20、20-30....等10个区间，程序中生成随机数之后进行区间判断，然后选取所在区间的小（大）值作为rowkey的盐。

③哈希

这里问题来了！！！！刚才我已经提到过一个问题就是添加随机数或者其他方式仅仅是避免写入和读出的热点，并没有将相关的数据放在一起！！！而哈希的方式的出现一定程度上解决了这个问题，首先将用户uuid+时间这个组合求hash然后模除以regionserver数量得到的值作为随机数放在rowkey前面，这样的话相同用户的同一天的数据将会放到一个regionserver中。这样做的方式很好的平衡了随机性和相似数据在一起的特性（rowkey有序性），如果不用随机数那么相似的数据就会完全在一起，如果用随机数，虽然相似数据在一起的几率小了，但是负载均衡性却提升了。

④反转

如手机号这种，前7位大多很相似，但是后四位比较随机，将手机号反转后就可以实现负载均衡性质，同时也保证的同一个用户的数据会存放在一起。

总结：我觉得最好的是采用(唯一标识+时间)取哈希再对regionserver取模的方式比较好！

## es部分

1. 说一说你对es的了解？

es是java语言开发，基于lucene的索引和搜索功能的近乎实时、高可用、分布式的全文检索引擎。可以由多个node组成集群作为一个独立的cluster运行，一个索引的数据保存在多个shard中，通过shard分片来达到数据的分布式存储和检索、复杂均衡的目的，每个shard都是一个lucene实例，也就是每个shard都有自己独立的待排索引，用于检索该shard的数据，有独立的（其实这里说的不是特别的准确，准确来说应该是segment，每个segment是lucene的实例，有独立的索引用于检索，而shard可以认为是大的segment，是多个小segment合并后形成的）。每个shard都会有replication副本，目的是提高吞吐量和高可用性。shard分片分主分片和副本分片，主分片用于写和读，副本只负责复杂均衡的读。

1. es的两个关键点是什么，分别说一说索引和压缩？

其实告诉我们了es的关键点就是强大的索引能力！！那么es的倒排索引到底牛逼在哪里？尝试跟B+树索引比较一下：

我们知道B+树的查询时logn的，并且他的插入一般不需要移动全部数据，而且B+数叶子结点的数组利用连续空间存储多个记录，方便一次性预读取。

es是通过倒排索引的方式（关于倒排索引等会还会继续说），为文档的每个term建立倒排索引，并且对这些terms进行排序，这样查找起来也是log你的，那么跟B+树效率没有区别啊，甚至可能比B+树还慢，因为占内存啊，B+数非叶子结点不存储数据的！

然而ES采用FST数据结构进一步加速term的查找速度，类似于trie树，将terms进行二次索引，trie树是根据字母的先后循序进行索引的，这样做可以大大节省内存，FST不仅利用了字符串前缀，还利用了后缀，大大降低了内存占用！与此同时类似trie数的查找时间复杂度是O(l(str))，基本上是O(1)。

ES的索引最终目的是除了进行倒排，还有一个目的是尽可能将索引放进内存。

1. 读写流程详细说一下？

写操作：

①协调节点默认使用文档ID作为哈希取模shard=hash(document\_id)%(num\_of\_primary\_shards)来确定把该文档储存到那个主分片上

②当那个主分片所在节点收到写请求之后，会首先写入内存的buffer中，并且将这一操作写入translog文件中，根据WAL的特点应该是先写log再写入内存，此时数据检索不到。

③es默认每隔1秒（可以修改）reflush到磁盘——注意根据操作系统的特性，写入的数据首先会进行filesystem的buffer，而不是真正的吸入磁盘，这样做的目的是减少寻道的随机写入。这os的buffer中实际上我们就可以认为是写入磁盘了，所以是可以被检索到的，另一方面，这一部分在os buffer中的数据已经完成了倒排索引工作！并且还有自己的名字segment！这个segement我们可以认为是已经写入磁盘的小lucene实例！只不过存在系统断电、宕机等缓冲区丢失的问题，而这一个问题由translog解决。

④那么什么时候真正被写入磁盘呢————flush操作，触发时机：默认30min或者translog大于512M的时候，这时候会将缓冲中的segments写入到磁盘，同时translog将被删除，并创建新的translog！

⑤然而translog也并不是非常安全的，你懂的即使是往磁盘写，操作系统的缓冲也可能导致translog丢失，所以es每5秒把translog写入磁盘。

⑥大量segment存在的话每次检索都会遍历这个分片的所有segement，所以es自动合并segments，合并方式是相似大小的进行合并出新的大的segment，旧的就删除

更新和删除操作：

注意：es的索引时不能进行修改的，所以更新和删除并不是在原有的segment和索引上操作的

每个segment都有对应的.del文件，用来记录删除的文件，每当用户发出删除请求时，文档并没有真正的删除，索引也没有变而是在.del文件中标记了该文档已被删除，因此被删除的文档依然可以检索到但是返回时被过滤掉了。每次当segement合并的时候，那么被标记删除的文档才会被真正删除。

对于更新其实也是一样，需要注意的时候每个文档都有自己的版本号，而更新就是利用了这个版本号，当有更新请求的时候在.del文件中加入该文档的id和版本号。并且重新插入一个新版本的一样id的文档，新旧文档都会被检索到，只不过就版本号会被过滤掉。

检索：

检索过程分为两个阶段：query阶段和fetch阶段。

①query阶段：协调节点会广播到所有分片（主或者副本）——注意与插入的区别，每个分片在本地执行搜索并构建一个客户端要求的大小size结果集。并且加入到分片的优先队列，位置时从from到size。然而每个分片只会返回一个轻量级的结果集——文档id+排序值。

②query阶段：协调节点会合并所有分片的结果，也就是全局排序，得到最终想要的哪些文档。

③fetch阶段：根据query阶段得到的所需结果集，知道了向哪些分片进行get请求，协调节点等到所有结果被拉取过来之后返回给客户端

注意：每个分片给出的文档排序是不准确的！因为每个shard是本地进行的排序，根据的是本地的数据进行排序的（打分），并不是全局的，所以存在一定弊端。当然也存在另一种查询方式就是DFS query and fetch，这种方式在查询之前先询问全局的文档和词频率，进而每个shard会给出准确的排序（平分），但是这种方式效率低。

1. ES是怎么进行并发读写的？说白了就是并发情况下如何保证读写一致性？和mysql有什么区别？

我们知道mysql的innodb引擎是通过行锁+MVCC的方式（Repeatable-Read隔离机制）来实现高并发下的读写一致性的。多版本并发控制技术，目的是实现非锁定读，也就是加了写锁之后还可以进行读。可以认为是乐观锁！

现在有点明白为什么MVCC是针对repeatable-read隔离条件下的应用了，因为如何是readcommited条件下，读锁根本不用管读写的一致性！

ES同样采用的是乐观锁，类似于多版本控制的方式。只不过比mysql的mvcc要稍微繁琐一点，为什么这么说：

ES的乐观锁方案是根据版本号进行标记，加入线程A操作将变量a-1=99，此时线程B在A改变之前读到了a=100，B也想a-1=99，此时B会它拿到的a的版本version=1是否与es中version=2相等，显然A已经改变了a的值，所以不相等，此时B会重新获取a的值=99，再次判断版本号是否相等只有相等时才会改变这个值。这个过程要持续不断的去请求+判断，而mysql并不是这样，如果存在两个修改操作将会等待阻塞，如果是读的话无所谓了就。我们可以看出es类似于非阻塞IO，mysql类似于阻塞IO，es也很类似与JDK5中的原子操作类atomic利用CAS（compare and set）实现基本类型数据的原子操作，jdk8进行了优化，采用分段CAS方式降低非阻塞IO的无限循环请求。

注意：：然而存在一个问题，就是我们知道读取请求的时候协调节点将读请求路由到主/副本shard上，如果存在副本shard此时并没有将其同步过来时就会出现读写不一致的情况！es是如何解决这个问题的？可以通过设置replication为sync同步来解决，就是说只有当主和副本shard都写成功后才会进行该查询请求，否则阻塞。如果设置replication为async异步同步的话，可以通过客户端设置\_preferance=promary的方式直接请求主shard来获取最新的数据！

另一方面对于写请求：es通过一致性配置quorum/one/all（默认是quorum），只有当大多数分片可用时可允许写操作，当然这种方式也存在一定问题，比如网络原因导致写入副本失败，这种情况下es会重建这个shard。

1. es的主节点选举是如何实现的、如何避免脑裂？

我们已经知道namenode借助zk的分布式锁的方式、kafka借助zk节点存储的ISR来实现partition的leader选举（普通选举、脏选举、禁止脏选举）、zk自己的leader是通过投票超过半数票的方式进行选举的、redis的master选举是借助其他master的投票，超过半数支持票的方式进行选举的。那么es的master是如何选举的呢？

在节点配置的时候会配置master：true才可以参与选举！

选举：

①es的选主是ZenDsicovery模块负责，主要包括ping（节点之间通过RPC来发现彼此）和Unicast（单播模块包含一个主机列表以控制哪些节点需要ping通）

②对所有可以称为master的节点根据nodeId字典排序，每次选举每个节点都把自己所知道的节点排一次序然后选出第一个（第0位）节点暂且认为它是master

③如果对某个节点的投票数叨叨一定值（可以成为master的n/2+1）并且该节点自己也选择自己，那这个节点就是master，否则再次选举。

那么什么是脑裂？学习hadoop的namenode失败迁移的时候就知道脑裂的问题，说白了就是发现有一个节点不行了就把另一个备用的切换为主，然而此时原来的主活过来了，那么现在客户端不知该向哪个主发送请求，这还不是关键，关键是当发生写、修改、删除请求的时候，就会出现不一致的问题，脑裂了。

而es的脑裂并不是说主服务器恢复而是指es也是根据超过半数投票的规则进行选举的（然而es并不是严格的半数票原则，而是通过设置，我们知道zk的半数原则是默认的n/2+1而es是可以配置的！如果这个配置不合理如有10台可以称为master，这个参数设置为10/2 + 1=6的话，完全没有问题，6票选中的节点会称为mater，但是如果这个参数设为10/2=5,的话，那么就会出现两个组都是5的投票，如果恰好这两组分别选择了不同的两个node，那么这两个node都将会成为master，那么现在就出现脑裂了）

es通过设置discovery.zen.minimum\_master\_nodes来防止脑裂，必须要设置可以称为master/2 + 1。

注意：集群模式下的master选举是对集群节点数有要求的，比如说zk最少是3个，因为如果zk是两个，那么死掉一个之后1 小于2/2+1，如果是三台死掉一台，2 =3/2+1所以可以选举，此时es也会出现这个问题，难道也要像zk一样必须要超过3台吗？答案是否定的，因为es是可以设置node.master=false，假如现在备用master的node有两台，可以把其中一台node.master设置为false，不然的话当master宕机之后根据配置3/2 +1=2，必须要两个一起存活才能进行master选举，假如在master宕机之前有一个备用master就已经宕机了，那么此时如果主master死了，根据配置3/2 +1=2，然而现在备用的master只有1个了，难道就不进行选举，不进行失败转移了吗？所以...

1. 说一说倒排索引吧？

通俗来讲：倒排索引就是词典和文章的映射关系。相反与一篇文章包含了哪些词，而是从词term触发，记载了在哪些文档中出现过，由两部分组成：词典和倒排表。

而这个倒排索引就是文档的索引，那么如何提升term在词典中的定位就是一个非常重要、非常需要优化的地方，如果默认情况下（排序过），查找速度是logn，这个速度怎么说，也就是比O（1）差一点而已，那么到底能不能达到O(1)呢？lucene3.0以前使用跳表skiplist来存储term dictionary，lucene4.0以后底层大量使用FST（finite state transducer）数据结构。

①空间占用下：目的是将term dictionary加载到内存，通过词典中单词的前缀和后缀的重复使用，注意postinglist不加载到内存。

②查询速度快O(len(str))的查询时间，可以认为是O(1)

注意：上面提到过term的倒排索引在内存中的存储结构并不是trie树，而是类似trie树的FST，那么FST是什么样的数据结构呢？可以认为它是一种压缩。

trie树是利用了公共前缀，而FST不仅利用前缀还利用公共后缀。将所有term加载到内存然后找到term之后返回要查询的term在term dictionary中的offset，然后直接获取到postinglist。

从这里可以看出为什么es的索引查询速度要快于mysql了，当表的索引不能全部加入到内存时就需要多次磁盘io，每次耗时大约10ms，而es的倒排索引由于term全部在内存中，并且以O(len(str))的速度查找，然后只需要进行一次磁盘io去倒排表中找postinglist即可。但是快是快但是耗内存、而且索引的更新不容易！