数据库（1）

- 数据库概论
  - SQL语言(内外连接，子查询，分组，聚集，嵌套，逻辑)
  - MySQL索引方法？索引的优化?
  - InnoDB与MyISAM区别?
  - 事务的ACID
  - 事务的四个隔离级别
  - 查询优化(从索引上优化，从SQL语言上优化)
  - B-与B+树区别?
  - MySQL的联合索引(又称多列索引)是什么？生效的条件?
  - 分库分表
- 数据理论基础
  - 基本概念
    - TPS：每秒并发事务数
    - QPS：每秒查询数
    - OLTP：on-line事务处理，特点是大量短事务，高并发，如关系数据库
    - OLAP：on-line分析处理，如数据分析
    - LSN：事务日志序列号
    - 脏页：内存中新写入的、未被刷新到磁盘的数据
    - 数据库实例：
      - 一组后台进程/线程以及一个共享内存区
      - 就算没有磁盘存储，数据库实例也能存在
      - 数据库实例内可以在内存中维护非持久化数据
      - 数据库可以由多个实例装载和打开
      - 实例在其整个生存期中最多能装载和打开一个数据库
  - 关系代数
    - 集合运算
      - 并
      - 交
      - 差
      - 笛卡儿积
    - 关系运算
      - 选择
      - 投影
      - 连接
      - 除
  - 数据库范式
    - 1NF
    - 2NF

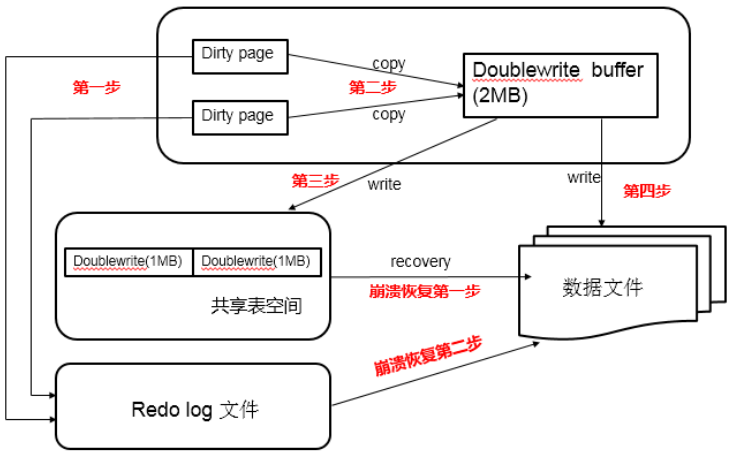
- 3NF
- BCNF
- 4NF

- MySQL
  - MySQL特性整理
    - MySQL5.7特性
      - InnoDB存储引擎的增强
        - Onlie DDL(alter table)
          - 在执行某些DDL操作时，不影响生产上的查询，以及DML操作
          - 创建、删除二级索引不需要创建临时表、复制数据
        - innodb_buffer_pool Online change
          - 引入了chunk（默认128M）来在线调整innodb_buffer_pool大小
          - buffer pool以innodb_buffer_pool_chunk_size为单位动态扩容和缩小
        - innodb_buffer_pool dump和load增强
        - InnoDB临时表优化
          - 临时表不再记录redo log
          - 临时表的数据形成了独立的表空间，不存放在ibdata1里了
        - page clean的效率提升
          - innod_db_page_cleaners可以指定page cleaner线程数量
        - undo log自动清除
          - innodb_undo_log_truncate默认关闭，可开启
          - 开启后，当undo log大小超过innodb_max_undo_log_size制定的最大值时，就会自动清除undo log
      - 其他方面增强
        - sys schema功能增强
          - 引入sys库，包含了一系列的视图、函数、存储过程，主要数据来源performance_schema，方便我们诊断问题
        - 复制功能增强
        - 设置SQL查询超时
          - max_execution_time参数设置SQL最大执行时间，方式长时间的SQL执行
        - 执行计划增强
          - 支持查看运行时SQL的执行计划
          - show full proccesslist查到的线程号可以让explain for connection 线程号； 查看
        - 优化器的提升
          - in子查询支持index range scan
          - union all不再产生临时表
    - MySQL8.0特性
      - InnoDB存储引擎的增强
        - 新的数据字典

- 新增了事务型的数据字典，用来存储数据库对象信息
- 之前，字典数据是存储在元数据文件和非事务型表中的
- 原子DDL
  - 数据字典的更新，存储引擎操作，写二进制日志结合成了一个事务
- 扩展select _____ for update
  - 新子句
    - no wait：无法获取到锁时直接返回错误，而不是等待
    - skip locked：忽略那些已经被其他session占有行锁的记录
  - 加入of ___ 指定要锁定的表
- 新增了动态配置项innodb_deadlock_detect
  - 用来禁用死锁检查，因为在高并发系统中，当大量线程等待同一个锁时，死锁检查会大大拖慢数据库
- 其他方面的增强
  - 持久化全局参数修改
    - 用特定的语法set persist来设定任意可动态修改的全局变量
    - mysqld-auto.cnf存放持久化的全局变量优先级高于my.cnf
    - persisted_globals_load参数决定启动时是否载入mysqld-auto.cnf
  - 添加了SQL角色功能，实现权限控制
  - 优化器的提升
    - 隐藏索引
      - 优化器可以忽略隐藏索引，但隐藏索引是被正常维护的，作用是用来测试无效索引
      - 删除某索引之前，可以先设置为隐藏索引，确定对系统没有影响后再删除，以防删掉后再次重建
    - 降序索引，可以对索引定义 DESC
      - 之前，索引可以被反序扫描，但影响性能，而降序索引就可以高效的完成
  - 取消Query Cache

- 基本
  - 安装和基本命令
    - 安装过程和配置
      - 启动Mysql服务：mysqld_safe --defaults-file=/etc/my.cnf &
        - mysqld_safe脚本会在启动MySQL服务器后继续监控其运行情况，并在其死机时重新启动它
        - 可以选择配置参数文件启动和默认启动，参数过滤等，安全性提高
    - 基本命令
      - show variables like '%___%'：查看参数
      - show table status(like '%___%' \G)：获取表基础信息
      - show create table ___\G：查看建表语句
      - desc ___：查看表结构
      - show index from ___：查看当前表下索引情况

        - show full processlist：查看连接情况
    - Query Cache
        - 它只缓存静态数据，一但数据发生变化，经常读写，就成了鸡肋
        - query_cache_size：可能会有一些缓存数据，建议清空以提高TPS
        - query_cache_type：在5.6之后默认为关闭
        - MySQL8.0之前的版本建议关闭
- 基准测试
    - sysbench
- 存储引擎：基于表，而不是数据库
    - 各版本区别
        - InnoDB：
            - 支持外键定义
            - 支持事务
            - 行锁
            - MVCC
            - 高并发
            - 数据文件都存储在.Ibd文件
            - 应用于OLTP业务
        - MyISAM：
            - 不支持外键定义
            - 不支持事务
            - 表锁
            - 低并发
            - 数据文件存储在.MYD文件，索引文件存储在.MYI文件
            - 只缓存索引
            - 应用于OLAP、ETL业务
        - TokuDB：
            - 支持事务
            - 支持压缩
            - 高速写入
            - 在线DDL
            - 不产生索引碎片
            - 应用于海量数据存储
        - MraiaDB columnstore：
            - 列式存储
            - 高压缩
            - 应用于OLAP业务
        - Blackhole：
            - 不存储数据，数据写入只写binlog
            - 常用来做binlog转储，或测试
    - InnoDB

- **InnoDB特性**
  - 插入缓冲(change buffer)
    - 把普通索引上的DML操作从随机IO变为顺序IO，提高IO效率
    - 参数：
      - innodb_change_buffer_max_size
        - 占innodb_buffer_pool的最大比例，默认25%。建议跳成50
      - innodb_change_buffering
        - change buffer的类型
        - 默认为all：缓冲全部insert、delete标记操作、和purges(物理删除)操作
    - 原理
      - 先判断插入的普通索引是否在缓存池中，如果在就直接插入
      - 如果不在则就放到change buffer中，然后进行change buffer和普通索引的合并操作
      - 可以将多个插入和并到一个操作中，**提高了普通索引的插入性能**
  - 两次写(double write)
    - 原理：操作系统并不能保障一个数据页操作的原子性
    - 参数：innodb_doublewrite，默认为1开启
    - 作用
      - 保证写入的安全性，牺牲一定写性能
      - 防止数据库实例宕机时，innoDB发生部分页写（partial page write）问题，即在一个页中写入一半时数据库宕机，导致写操作失效
      - 数据库实例崩溃时，redo log不能恢复内存中的数据，它记录的是对页的物理操作，所以我们需要页的副本
      - 如果实例宕机了，就可以先通过副本把原来的页还原出来，再通过redo log进行恢复、重做
    - 原理
      -


      - InnoDB缓冲池中刷出的脏页在被写入数据文件之前，先拷贝至内存中的两次写缓冲区**double write buffer**，大小为2M

        - 然后从double write buffer分两次。每次将1M大小的数据写入磁盘**共享表空间**
        - 最后再将double write buffer写入**数据文件**
    - 自适应哈希索引（AHI）
        - InnoDB引擎可以监控索引的搜索，如果InnoDB注意到查询可以通过建立哈希索引得到优化，那么它就自动完成这件事
        - 自适应哈希搜索系统是分区的，每个索引都会绑定到一个特殊的分区上
        - 作用
            - InnoDB会自动根据访问频率和模式自动地为某些热点页建立哈希索引
            - 它是通过缓冲池中的B+树构造而来，且不需要对整个表建立哈希索引
            - 启用自适应哈希索引后，读和写性能可以提高2倍，对于辅助索引的连接操作，性能可以提高5倍
            -
        - 要求
            - 对一个页的连续访问模式是一样的
                - 如下操作(a,b为联合索引)如果交替进行，那么InnoDB不会构造AHI
                    - WHERE a=xxx
                    - WHERE a=xxx and b=xxx
            - 通过相同模式访问100次
            - 页通过相同模式被访问了N次，N=页中记录/16
        - 参数
            - innodb_adaptive_hash_index，默认开启
            - innodb_adaptive_hash_index_parts，控制自适应哈希搜索系统的分区数。默认为8个
    - 刷新邻接页
        - 参数：innodb_flush_neighbors
        - 当刷新一个脏页时，InnoDB会检查该页所在区的所有页是否也为脏页，如果是，那么也一起刷新
        - 但对高IOPS性能的硬盘，如SSD，建议关闭
    - 异步IO
- 存储结构
    -

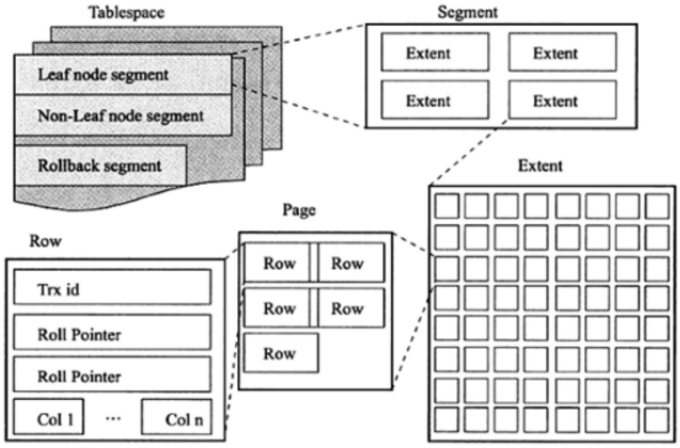
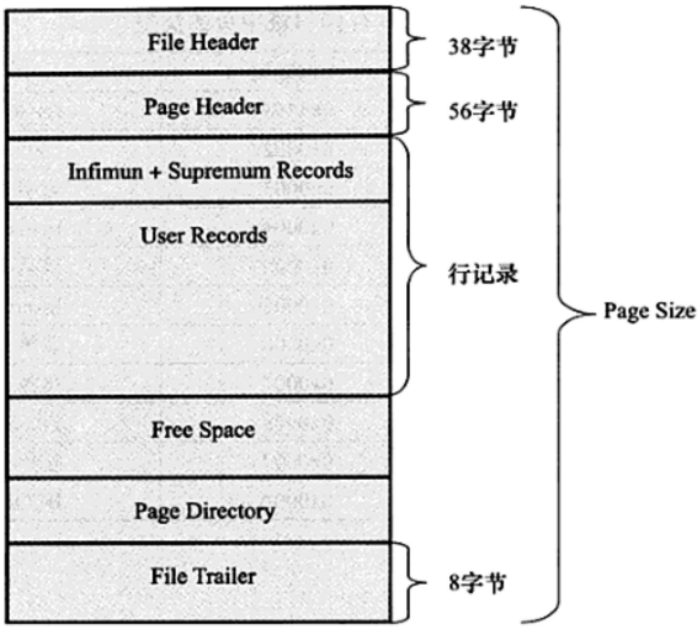


图 4-1 InnoDB 逻辑存储结构

- 表空间
  - 表空间是Innodb存储引擎逻辑的最高层，所有的数据都存放在表空间中
  - 共享（系统）表空间ibdata1
    - 存储所有信息以及回滚（undo）段信息
    - 默认情况下，Innodb存储引擎有一个共享表空间ibdata1,即所有数据都存放在这个表空间中内
    - innodb_data_file_path负责定义系统表空间路径、初始化大小、自动扩展策略，建议将自动扩展大小调整为1G，因为12M的默认大小会使高并发事务受到影响
    - 无法实现在线回收空间，想要回收它必须把全部innoDB表中的数据备份删除原表，再把数据导入到新表中。
    - 统计分析、日志类系统不适合使用共享表空间
  - 独立表空间
    - 存储表对应的索引，插入缓冲，B+树数据信息
    - **默认开启**，或设置innodb_file_per_table=1开启
    - 每一个表使用自己的表空间，而不用将全部信息存储在ibdata1中
    - 使用独立表空间，容易实现表空间的转移，但每个文件都有.frm和.ibd两个文件描述符，单表增长过快容易导致性能问题
    - 但综合考虑，独立表空间效率、性能会比共享表空间高一点
  - 临时表空间
  - 通用表空间
- 段
  - 表空间由段组成，常见的段有数据段、索引段、回滚段等
  - 段由N个区和32个零散的页组成
  - 数据段即为B+树的叶子结点，索引段即为B+树的非叶子结点
  - 创建一个索引一般就会创建两个段
  - 在InnoDB存储引擎中对段的管理都是由引擎自身所完成，DBA不能也没必要对其进行控制
- 区

        - 区是由连续页组成的空间，在任何情况下每个区的大小都为1MB。为了保证区中页的连续性，InnoDB存储引擎一次从磁盘申请4~5个区
        - **默认情况**下，InnoDB存储引擎页的大小为16KB，一个区中一共64个连续的页
    - 页
        - 默认16KB，InnoDB磁盘管理的最小单位
        - 页大小参数innodb_page_size



        - 一个page默认预留1/16的空间用于更新数据，最少可以存2行数据
        - 常见的页类型有：数据页，undo页，系统页，事务数据页，插入缓冲位图页，插入缓冲空闲列表页
    - 行
        - 页里又存放着记录行的信息，InnoDB存储引擎是面向列的，也就是说数据按行存放。每个页存放的行记录也是有硬性定义的，最多允许存放16KB/2-200=7992行记录
        - Innodb文件格式有Antelope和Brracuda两种
        - 行记录的格式
            - Antelope：compact、redundant
            - Brracuda：dynamic、compressed
            - 5.7版本**默认使用dynamic行记录格式**
            - 行溢出：将一条记录中的某些数据存储在真正的数据页面之外，拆分到多个页进行存储
            - compact行格式，溢出的列只占768个前缀字节
            - dynamic实际采用的数据都存放在溢出的页中，数据页只有前20个字节的指针，针对溢出列所在新页的利用率更高
- MySQL内存结构
    - SGA(系统全局区)
        - innodb_buffer_pool
            - 缓存InnoDB表数据、索引、插入缓冲、数据字典

- innodb_log_buffer
  - 事务在内存中的缓存，即redo log buffer大小
- Query Cache
  - 高速查询缓存，但只缓存静态数据，建议关闭
- key_buffer_size
  - 只用于MyISAM表
- innodb_additional_mem_pool_size
  - 保存数据字典信息和其他数据结构的内存池大小
  - 5.7.4被移除

- PGA(程序缓存区)
  - sort_buffer_size
    - 用于SQL语句在内存中的临时排序
  - join_buffer_size
    - 表连接使用，用于**BKA**
  - read_rnd_buffer_size
    - Mysql随机读缓冲区大小，用于做**mrr**
- 其他
  - tmp_table_size
    - SQL语句在排序或分组时没有用到索引，就会使用临时表空间
  - max_heap_table_size
    - 管理heap、memory存储引擎表
  - 以上两个值建议设置为一样大小
  - default_tmp_storage_engine
    - 临时表默认存储引擎（5.7版本默认InnoDB）
  - internal_tmp_disk_storage_engine
    - 磁盘上临时表管理，由(CREATE TEMPPRARY TABLE)管理

- InnoDB Buffer状态及其链表结构
  - page是InnoDB磁盘I/O最小单位，数据存放到page中，对应到内存中就是一个buffer，它由三种状态：
    - free buffer
      - buffer从未被使用，数据库繁忙的情况下基本不存在
    - clean buffer
      - 内存中buffer中的数据和磁盘page一致
    - dirty buffer
      - 内存中新写入的数据还没刷新到磁盘，跟磁盘数据不一致
  - buffer由双向链表组织起来
    - free list
      - 其上的节点都是free buffer，如果需要从数据库中将新的page调入内存，直接从上获取即可，如果不够用了就从flush list或和lru list淘汰一定的节点

- lru list
  - 它会把那些与磁盘数据一致，并且最少被使用的buffer串联起来，释放出free buffer
- flush list
  - 将dirty buffer串联起来，方便刷新线程把数据刷到磁盘
  - 推进check Lsn，使实例崩溃之后，可以快速恢复
  - 使用lru规则，将最少被更新的数据刷到磁盘后，释放出free buffer

- InnoDB内存刷新机制
  - MySQL，执行DML语句总是先写日志，再写数据文件
  - 刷新线程
    - master thread
      - 后台线程的主线程，优先级最高
      - 内部有四个循环：主循环loop，后台循环background loop，刷新循环flush loop和暂停循环suspend loop，根据数据运行状态在四种循环之间切换
      - loop循环
        - 每1s操作
          - 日志刷新到磁盘，即使事务未被提交（**redo log thread**）
          - 刷新脏页到磁盘（**page cleaner thread**）
          - 执行合并插入缓冲的操作（**change buffer thread**）
          - 产生checkpoint
          - 清除无用的table cache
          - 如果当前没有用户活动，就可能切换到background loop
        - 每10s操作
          - 日志刷新到磁盘，即使事务未被提交
          - 执行合并插入缓冲的操作
          - 刷新脏页到磁盘
          - 产生checkpoint
          - 删除无用undo页（**purge thread**）
    - read/write thread
      - 读写请求线程
    - change buffer thread
      - 将插入缓冲内容刷新到磁盘
    - redo log thread
      - 将重做日志内容刷新到磁盘
    - page cleaner thread
      - 脏页刷新线程
    - purge thread

- 负责删除无用undo页
- 进行DML语句都会生成undo，所以需要定期清理

- checkpoint thread
  - redo log发生切换或者文件快写满时，将脏页刷新到磁盘
  - 还可以确保redo log刷新到磁盘

- redo log的刷新机制
  - redo log（重做日志文件）用于记录事务发起后的DML和DDL SQL语句，物理日志（作用于文件，文件损坏就失效），记录的是修改后的表数据，不管事务是否提交都记录下来
  - 用于异常宕机或者介质故障后的数据恢复，默认情况至少有两个redo log文件
  - 刷新条件
    - 通过**innodb_flush_log_at_trx_commit**参数决定
      - 0，性能最好，但不安全
        - redo log thread每隔1秒会将redo log buffer中的数据写入redo log，同时刷新磁盘
        - 但此参数会让每次事务提交时日志缓冲中的数据不会被写入redo log
      - 1，安全性最高，性能最差
        - 每次事务提交时触发刷新，并将数据刷新到磁盘
        - 保证在主机断电，系统宕机下不会损失任何已提交数据
      - 2，介于两者之间
        - 每次事务提交时触发刷新，但数据并不刷新到磁盘
    - master thread：每秒进行刷新
    - redo log buffer：超过其一半时会触发刷新

- bin log的刷新机制
  - bin log(二进制日志文件)记录commit完毕后的DML和DDL SQL语句，逻辑日志，记录所有数据的改变信息
  - 用于恢复数据，主从复制搭建
  - 只要binlog写入完成，那么主从复制环境中，都会正常完成事务
  - 刷新条件
    - 由**sync_binlog**参数决定
      - 值为0，就让文件系统自行决定什么时候做同步
      - 值为n，则每n次事务提交之后MySQL进行一次fsync之类的磁盘同步指令将binlog_cache中的数据强制写入磁盘
      - 为保证数据安全性，可以将该参数设置为1

- 数据库文件
  - 参数文件my.cnf
    - 存放和设置所有全局变量的配置文件
  - 错误日志文件log_error
    - error.log文件

- 二进制日志log_bin
  - 格式（bin_log_format）
    - statement：基于操作的SQL语句记录
      - 缺点：在某些情况下导致master-slave中的数据不一致
    - row(推荐格式)：基于行的变更情况记录
      - 不记录每条记录的上下文信息，只记录行变更前的样子和变更后的内容
      - 任何情况下都可以被复制
      - 缺点：会产生大量日志
    - mixed：混合前两种方式
  - 作用
    - 完成主从复制，主服务器把所有修改数据的操作记录到binlog中
    - 进行恢复操作，使用mysqlbinlog命令，实现基于时间点和位置的恢复
- 慢查询日志（log_slow）
  - 把超过long_query_time的日志记录进slow.log文件
  - 查看慢查询日志的方式
    - mysqldumpslow
    - percona-toolkit：pt-query-digest
    - druid
- 全量日志（log_output）
  - 记录数据库所有操作的SQL语句，log_output参数默认关闭
- 审计日志
  - 实时记录数据库的活动，能实现对数据库操作的监控
- 中继日志（从库的log_relay）
  - 从库会读取该日志内容并应用
- Pid文件（.pid）
- Socket文件（mysql.sock）
- 表结构文件（.frm文件）
  - 8.0版本取消了它，Mysql8.0用InnoDB存储引擎表DDL语句操作的原子性
- InnoDB存储引擎文件
  - redo日志文件
    - 用于记录事务操作变化，记录的是数据被修改后的值，见前面刷新机制
  - undo日志文件（innodb_undo_log）
    - 对记录做变更操作时，记录变更前的旧数据
    - 默认存放到系统表空间ibdata1中，5.6开始可使用独立的undo表空间
    - 主要参数
      - innodb_undo_logs
        - 回滚段默认为128个
        - 每个undo log segment最多存放1024个事务
      - innodb_undo_tablespaces
        - 推荐至少设置为2

        - undo log某个表空间被truncate时，保证由可用的undo log tablespace能提供使用
      - innodb_undo_log_truncate
        - 默认关闭，truncate后的表默认恢复为10M
        - 当阈值超过innodb_max_undo_log_size（默认1G）时truncate undo logs

- 表
  - 建表原则
    - 禁止中文字段
    - 禁止字符型主键
    - 禁止无主键或唯一索引的表出现
  - 数据类型的选择
    - 原则：选择最小，最合适的类型
    - 整型
      - int unsigned做主键基本满足业务就不用bigint（对应long类型）
      - int（n）中的n只代表显示宽度，不代表字节数
    - 浮点型
      - 不推荐使用float和double，因为不精确
      - 推荐decimal(M，N)
        - M代表整数和小数总长度，N代表小数保留位数（四舍五入）
        - 插入的数字整数部分不能超过M - N位，否则报错
        - 但它计算时仍是转换为浮点数运算，而且也会出现四舍五入的情况，导致计算不准确
    - 时间类型
      - date/time：3字节
      - timestamp：4字节
      - datetime：5字节
      - 也可以用int来存时间：通过两个函数转换：unix_timestamp和from_unixtime
    - 字符串类型
      - char：定长字符（0-255字节）
        - 字符未超过指定指定位数，那它在后面补空格
      - varchar：变长字符（0-255/65535字节）
        - **varchar（n）：n代表字符**
        - 长度为0-255，用1字节存长度
        - 长度为256-65535，用2字节存长度
        - 还有1字节记录是否为null值
      - 推荐用int来存IP地址
        - inet_aton：IP地址转换为int类型
        - inet_ntoa：int类型转换为IP地址
  - 表碎片

- 删除数据时，文件大小没有减少，因为删除后数据文件遗留了大量的数据碎片
- 表碎片产生原因和导致的问题
  - MySQL使用delete删除数据的时候，只是将数据文件的标识位删除，也没有整理文件因此不会彻底释放空间
  - 由于碎片也占用了硬盘空间，所以读取效率方面比正常占用的空间要低很多
- 表碎片大小计算
  - 用show table status like'%___%'查看表信息
  - 碎片大小 = 数据总大小 - 实际表空间文件大小
  - 数据总大小 = data_length + index_length
  - 实际空间文件大小 = rows * avg_row_length
- 表碎片清理方法
  - 常规方法
    - alter table ___ engine=innodb
      - 这样会重新整理一遍全表数据，但会对整表加写锁
      - 但5.7版本后支持online DDL
    - 备份原表数据，然后删掉，重新导入到新表
  - 使用第三方工具
    - 如percona-toolkit里的pt-online-schema-change，但有了online DDL后就不需要了

- 表统计信息
  - MySQL information_schema数据库中的tables表记录了所有表的统计信息
  - 可以对tables表做查询操作并计算获得所需信息
  - show table status like'%___% 也可以查看部分信息

- 索引
  - MySQL的B+树索引
    - 关键字信息都出现在叶子节点中，叶子节点按关键字大小顺序连接形成双向链表结构
    - 聚集索引
      - 索引键值的逻辑顺序决定了表数据行的物理存储顺序
      - 索引叶子节点存储所有行记录信息
      - 一张表只能有一个聚集索引
      - 如果破坏了聚集索引的逻辑顺序，会导致物理顺序的变更，再次排序，这会耗费很多时间
    - 普通索引（二级索引，辅助索引）
      - 创建、删除二级索引不需要创建临时表、复制数据
      - 索引叶子节点只是保存自己**本身的键值**和**主键的值**，不保存行记录信息
      - 普通索引通过叶子节点上的**主键**来获取想要查找的行记录
  - 列的索引选择性的计算
    - 可以用count(distinct ____) / count(*) 计算出选择性
    - 选择性越高，代表重复值越少，越适合建索引

- ICP、MRR、BKA
  - ICP(index_condition_pushdown)：索引条件下推
    - Mysql使用索引从表中检索数据的一种优化方式
    - 如果where语句可以使用索引，那么Mysql会把where语句的过滤操作放到存储引擎层，**存储引擎通过索引过滤数据**
    - ICP能减少存储引擎层访问基表的次数和Server层访问存储引擎的次数
  - MRR(Multi-Range Read)
    - 参数
      - mrr
      - mrr_cost_based
        - 是否通过基于成本的算法来决定mrr开启，判断代价过高时就不会使用该项优化
        - on，表示自行判断，off表示强制开启
    - 作用
      - **将随机IO转换为顺序IO**
      - 在使用二级索引做范围扫描的过程中减少磁盘随机IO和减少主键索引的访问次数
    - 原理
      - 将通过普通索引找到的主键值集合存放在read_rnd_buffer中
      - 然后在read_rnd_buffer中对主键排序，再去访问表中数据，这样就成了顺序IO
  - BKA（Batched Key Access）
    - 参数
      - batched_key_access
        - 必须在开启mrr参数的基础上开启
    - 作用
      - 在被join表上有索引可以利用，那么就在行提交给被join的表之前，对这些行按照索引字段进行排序，因此减少了随机IO
      - 提高join的性能，在读取被join的表记录时使用顺序IO，按连接条件顺序比对数据
    - 原理
      - 对于多表join语句，Mysql**使用索引**访问第二个join表时，使用一个join buffer来收集第一个操作对象生成的相关列值
      - BKA构建好key（关键字）后，批量传给存储引擎层做索引查找
      - key是通过MRR接口作主键排序，再提交给存储引擎的，这样一来MRR使查询更高效
- 主键索引
  - 默认被创建为聚集索引
  - 主键必须唯一，非空，并且最好不要更改主键的值
  - 一定要保证该值是自增的，这样也能保证插入顺序也是自增的，不会导致重排序，提高了存取效率
- 唯一索引
  - 不允许有重复值的普通索引

    - 允许有null值
- 覆盖索引
    - 所建的索引覆盖了要查询的列
    - 覆盖索引是查询的数据列从索引中就能够取得，不必再回磁盘查询数据，大大减少了IO操作
    - 普通索引由于包含了主键的值，所以普通索引就覆盖了主键列，那么就可以直接得到主键的列值了
    - 使用覆盖索引不能有 * 号
- 前缀索引
    - 对于BLOB、TEXT、或者很长的VARCHAR类型的列，为他们的前几个字符（建立时指定字符数）建立的索引
    - 不能在ORDER BY或GROUP BY中使用前缀索引，也不能用作覆盖索引
    - 语法：alter table table_name add key(column_name(prefix_length));
- 联合索引
    - 有两个或两个以上的字段组成的索引
    - 使用时满足**最左前缀原则：查询语句可以使用索引中的一部分列,但必须从最左侧开始**
    - 一般把选择性高的列放在最前来建立联合索引
    - 如索引是key index (a,b,c). 可以支持a | a,b | a,b,c 三种组合进行查找，其实也就是建了这三个索引
    - 好处：可以从索引中直接获得列值，避免回表，减少磁盘IO
- 哈希索引
    - 使用哈希算法，将键值生成哈希值，只要一次哈希算法即可定位到相应位置，加快查找效率
    - 但哈希索引值能进行等值查询
- 合理使用索引（SQL索引优化）
    - **小心MySQL的隐式类型转换**，可能会导致索引失效，写SQL时要严格按照列的数据类型来写列值，比如：字符串类型是'123'而不是123
    - 尽量在程序端多一些判断，而不让数据库做各种运算，SQL中尽量避免or关键字，多列最好用union来代替
    - SQL语句优化步骤
        - 先看表的**数据类型**是否合理，是否满足数据类型越简单，越小的原则
        - 表中**碎片是否整理**，利用表信息进行计算来判断
        - 表的统计信息是否收集，统计信息准确，执行计划才可以帮助我们做优化
        - 查看**执行计划**，检查索引使用情况，如果没有用到索引，就考虑创建
        - 建立索引之前还要看字段的**索引选择性**，判断字段是否适合创建索引
        - 创建之后查看执行计划，对比结果，看是否提高查询效率
    - 适合建立索引的三个"经常"
        - 经常被用做查询条件的列（where语句后面）
        - 经常用于表连接的列
        - 经常排序和分组的列（order by、group by语句后面）
- 索引使用总结

- 索引的优点
  - 提高数据检索效率
  - 提高聚合函数效率
  - 提高排序效率
  - 使用覆盖索引可以避免回表
- 建立索引的四个不要
  - 选择性的的字段不要创建索引（如sex，status）
  - 很少查询的列不要创建索引（项目初期就决定好）
  - 大数据字段不要用索引
  - 不要使用NULL，最好加上NOT NULL（NULL会使运算更复杂，可以使用空字符串代替NULL）
- 使用不到索引的情况
  - 通过索引扫描的行记录超过表的30%，优化器就不会走索引，变为全表扫描
  - 联合索引中，第一个查询条件不是最左前缀
  - 联合索引中，第一个列使用范围查询，只能用到部分索引，有ICP（把查询条件过滤交给存储引擎）出现
  - 模糊查询条件列最左以%开始（可以考虑放子查询里面）
  - 两个单列索引，一个用于检索，一个用于排序，只能用到一个索引。考虑建立联合索引
  - 查询字段有索引，但使用了函数运算

- 事务（InnoDB）
  - ACID特性
    - 原子性(atomic)
      - 事务中所有操作要么都做要么都不做
    - 一致性(consistency)
      - 数据库中的数据在事务操作前和事务处理后，必须都满足业务规则约束
    - 隔离性(isolation)
      - 防止多个事务并发执行时由于交叉执行导致数据不一致
    - 持久性(durability)
      - 事务处理后，对数据的修改就是永久的
  - 事务语句
    - 隐式提交：执行DDL或再次开启事务时自动提交，DDL语句默认自带一个coommit
    - truncate和delete区别
      - truncate是DDL语句，delete是DML语句
      - truncate（DDL提交了事务）不能被回滚，delete能被回滚
      - truncate会清空表的自增ID属性，从1开始重记录，delete不会
  - 隔离级别
    - read uncommit(RU)
      - 一个事务读取到其他事务未提交的变化，读取到还没提交的事务的数据，叫做脏读

- read commit（RC）：不可重复读
  - 一个事务，只可以读取其他事务已经提交的数据变化，存在幻读，Oracle默认隔离级别
  - 不可重复读：一个事务读取到了其他事务对旧数据的修改（update/delete）
  - 幻读：在一个事务中读取到了其他事务新增的数据（insert）
- repeatable read（RR）
  - Mysql默认隔离级别
  - 每次读取事务开始前的数据，结果都是一致的
  - 一个事务开始后，直到事务结束，都可以看到事务开始时的数据，并一直不会发生变化，避免脏读，幻读，不可重复读
  - 如果想要读取到新增数据的信息，可以在查询语句后加上for update
- serializable
  - 每个读的数据行上都要加表级共享锁，写的数据行上都要加表排他锁，不适合并发场景

- MySQL锁
  - 锁分类
    - 读锁(共享锁，S锁)
      - 一个事务获取了一个数据行的锁，其他事务能获得该行对应的读锁，但不能获得写锁
      - 使用方式
        - 自动提交模式下的select查询语句，不需要加任何锁（一致性非锁定读）
        - 通过select ___ from lock in share mode在被读取的记录行或行的范围加上一个读锁，其他事务可读，但申请写锁会阻塞
    - 写锁（排他锁，X锁）
      - 一个事务获得了数据行的写锁，其它事务就不能获取改行其他的锁
      - 使用方式
        - 一般DML修改语句都会对行记录加写锁
        - 通过select ___ for update，对读取的数据加写锁
    - MDL锁（meta data lock）
      - 用于保证事务开启后的表中元数据信息
      - 一个会话开启了查询事务后，会自动获得一个MDL锁，其他会话就不能执行任何DDL操作，以保证数据的一致性
    - 意向锁
      - InnoDB的表级锁
      - 跟MDL类似，都是防止事务执行过程中执行了DDL操作，导致数据不一致
      - 意向共享锁（IS）
        - 给表中一个数据行加共享锁之前，必须先获得该表的IS锁
      - 意向排他锁
        - 给表中一个数据行加排他锁之前，必须先获得该表的IX锁

- InnoDB行级锁
  - 单个记录的锁（record lock）
    - 两事务更新同一行数据，其中一方会等待锁的释放（另一方commit）
    - **InnoDB的行记录锁是加载索引项上的，非索引项还是加表锁**
  - 间隙锁（gap lock）：RR隔离级别生效
    - 为避免幻读现象，RR隔离级别引入了间隙锁，它只锁定行记录数据的范围，不包含记录本身
    - RR隔离级别不允许在锁定的行记录范围内插入任何数据，从而避免幻读
  - 单记录锁和间隙锁的组合（next-key lock）
    - 当InnoDB**扫描索引记录时**，会先对选中的索引记录加上record lock，再对索引记录两边的间隙加上gap lock
- 锁等待和死锁
  - 锁等待
    - 一个事务必须等待另一个释放它的锁才能占用释放的资源，否则必须等待，等到超时就返回错误
    - innodb_lock_wait_timeout参数控制超时时间，单位是秒
  - 死锁
    - 多个事务争用资源而形成的相互等待锁释放的现象
    - InnoDB在出现死锁时，会回滚最后进入等待导致并死锁的那个事务
    - 降低数据库死锁发生的概率的方法
      - 多个程序并发存取多个表或记录，尽量约定以相同顺序访问表
      - 业务中尽量使用小事务，并及时提交或者回滚
      - 同一个事务中尽可能一次做到锁定所需要的全部资源
      - 对非常容易产生死锁的业务部分，可以尝试升级锁粒度（如表锁，减少因行锁导致的死锁发生的概率）
- 锁问题监控
  - 习惯通过show full processlist，show engine innodb status来判断事务中锁的问题
  - information_schema下的innodb_tx，Innodb_locks，innodb_lock_waits三张表可以帮助我们更好的分析事务中锁存在的问题

- 备份恢复
  - 冷备及恢复
    - 冷备是在数据库关闭时的备份，过程简单，速度相对快
    - 过程：复制整个数据目录到备份机或者磁盘上
    - 恢复：用已备份的目录替换源目录
  - 热备及恢复
    - 在数据库运行状态下备份，不影响业务进行
    - 逻辑备份（备份SQL语句，用SQL语句进行恢复操作）
      - mysqldump：可以备份表结构，表数据
        - Mysql自带，它先从buffe找到所需要备份的数据，如果buffer中没有就去磁盘的数据文件中查找并调回到buffer再备份，形成一个可编辑的备份文件

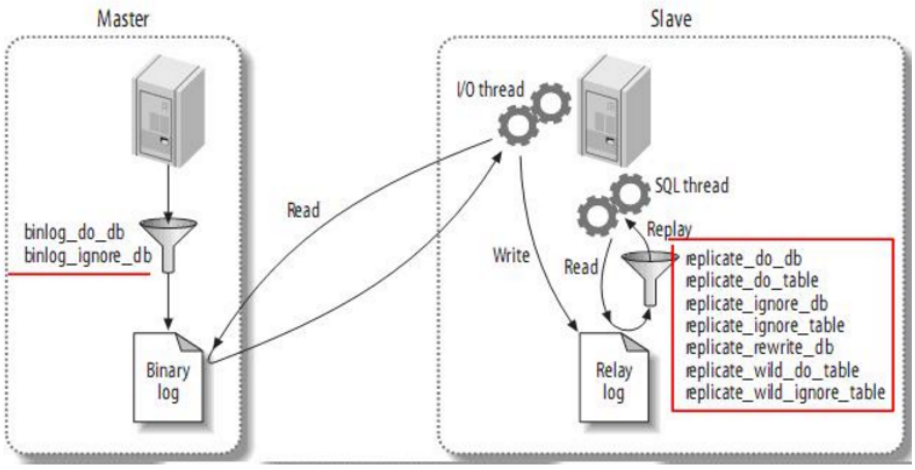
- mysqldump备份可能会遇到数据库性能抖动问题，出现性能急剧下降的现象
  - 因为它的备份过程，把**数据从磁盘中调回内存时**，可能把**内存中的热数据冲掉**，影响了业务访问
  - 5.7之后，新增innodb_buffer_pooldump_pct参数
    - 用它来控制buffer中转储活跃使用的innodb buffer pages的比例，默认值为25%
    - 只有当数据在1秒内再次被访问才能放到热区域，避免了热数据被冲走
- select___into outfile：只能备份表数据
  - 它的恢复过程非常快，比insert快很多(10条数据比insert快约10倍)
  - 备份操作：select _____ from _____ into outfile '/path/file_name[.sql]';
  - 恢复操作：load data infile '/path/file_name' into table _____;
- mydumper
  - 第三方工具，需下载
  - 优点
    - 多线程备份工具
    - 支持文件压缩功能
    - 支持多线程恢复
    - 保证数据一致性
    - 备份和恢复过程速度比mysqldump快
- 裸文件备份：在底层复制数据文件，不需要一条条执行SQL语句恢复，比逻辑备份快
  - XtraBackup
    - 目前不能对表结构文件和其他非事务表备份
    - 优点
      - 备份和恢复速度快，安全可靠
      - 备份过程不会锁表，不影响现有业务
    - 原理
      - 对InnoDB来说，XtraBackup是基于InnoDB的crash recovery进行备份的
      - crash recovery
        - InnoDB引擎的一个特点，当故障发生，重新启服务后，会自动完成恢复操作，将数据库恢复到之前一个正常状态
        - 恢复进程会完成两步
          - 第一步：检查redo日志，同步之前完成并提交的事务
          - 第二步：将undo日志中，未完成提交的事务，全部取消（回滚）
      - XtraBackup不锁表，一页页地去复制InnoDB的数据，但要保证数据的一致性，需要在备份恢复时使用crash recovery进行操作
      - XtraBackup有一个线程监视着redo log，在日志发生变化时（因为redo log的循环读写）就复制变化过的log pages，复

制完全部数据后停止复制redo log
- 恢复时使用 --apply-log参数触发crash recovery同步已经提交的事务，回滚未提交事务到数据文件，保证数据文件的一致性

- 备份方式(innobackupex)
  - 全量备份
    - 对一个数据库的全部数据进行备份
    - 恢复时要关掉数据库实例，移动文件，完成恢复
  - 增量备份：--incremental
    - 每一次备份都基于上一次的备份
    - 基于全备而言，第一次增备数据要基于上一次全备，之后每一次备份都基于上一次的增备，最终达到一致性的增备
    - 原理
      - 备份集中有xtrabackup_checkpoints文件，记录着备份完成时检查点的lsn
      - 进行新的增量备份时，xtrabackup会比较表空间中的每页的lsn是否大于上次备份完成时的lsn，如果大于则备份，并记录当前检查点的lsn
    - 恢复过程
      - 首先进行全备恢复校验，然后把增备文件恢复到全备文件中，注意这两操作要加--redo-only参数，只前滚已提交事务，不回滚未提交事务
      - 最后对整体全备进行恢复校验，这里要去掉--redo-only，回滚未提交事务
      - 然后关掉数据库实例，移动文件，完成恢复
  - 流式化备份
    - 可以不用备份到磁盘
    - xtrabackup使用--stream指定输出格式（tar或xbstream），还可以加管道符在后边指定压缩格式，最后 > 到备份文件
  - 远程备份
    - 使用流式化备份可以把备份文件传到远程备份机上（如ssh）,以使本地在磁盘空间不足时影响业务

- 表空间传输
  - 把一张表从一个数据库移到另一个数据库或者机器上
  - 使用条件
    - 1）5.6版本以上
    - 2）使用独立表空间（innodb_file_per_table）
    - 3）源库与目标库的page_size必须一致
    - **4）当表导出后，该表只能做只读操作**
  - 步骤
    - 在数据库A中，创建一个相同表结构的表
    - 在数据库A中，卸载待转移表的表空间(删除了__.idb文件)：alter table ___ discard tablespace;

        - 在数据库B中，导出转移表：flush table ___ for export;
        - 将数据库B中的__.ibd文件复制到数据库A中
        - 修改数据库A数据文件的系统权限：chown，并将数据库B的表解锁：unlock tables;（见条件4））
        - 数据库A中执行导入操作：alter table ___ import tablespace
    - 利用binlog2sql进行闪回
        - 当数据库遇到误删除、改错数据的情况可以进行快速恢复
        - binlog2sql工具能实现数据快速回滚，从binlog中提取SQL，并生成回滚SQL语句
        - binlog以event作为单位记录数据库变更的信息，闪回就是可以重现这些变化数据信息之前的操作
        - binlog2sql安装注意
            - 依赖：python-pip、PyMySQL、pythin-mysql-replication、wheel-argparse
            - 解压后在安装目录：pip install -r requirements.txt
        - binlog2sql.py注意事项
            - 保证mysql服务开启，离线无法分析binlog
            - binlog_format必须是row格式
            - DDL无法做到闪回，只能解析DML语句
    - binlog server
        - 自带的mysqlbinlog工具可以**把远程机器的日志备份到本地目录**
- 主从复制（主从同步）
    - 功能
        - 构建高可用集群的基础
        - 利用Mysql主从复制，可以实现实时灾备，让从库随时可以接管有故障的主库
        - 也可以让从库分担主库压力，做读写分离提供查询服务
        - 还可以让从库做一些特使SQL的统计任务，或是用从库做备份
    - 主从复制原理
        - 主库的工作线程为I/O dump thread，给从库I/O thread传送binlog日志



        - 过程
            - 主库把接收的SQL请求记录到自己的binlog日志中

        - 从库的IO thread去请求主库的binlog日志，并将得到的binlog写入到自己的relay log中
        - 利用SQL thread读取relay log中的SQL语句，并重做
- 异步复制（默认方式）
    - 主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理
    - 缺点：主库如果crash掉了，此时主库上已经提交的事务可能并没有传到从库。如果此时强行将从库提升为主库，可能导致新主库上的数据不完整
    - 搭建主从的必要条件
        - 主从库的server-id不一致
        - 主库开启binlog功能
        - 为了保证后期不出现数据不一致的情况，binlog格式要保证为row
    - 过程
        - 创建一个主从复制账号
        - 初始化数据，从主库中导出数据，导入到从库(用到备份恢复)
        - 从库执行主从复制命令change master to _____ master_log_file=____, master_log_pos=____;
        - 开启主从复制start slave
        - 查看状态：show slave status\G
    - 搭建主从的管理命令
        - show slave status\G：在从库上查看主从复制状态
        - show master status：查看主库的binlog和position，以及开启GTID模式下记录的GTID
        - change master to：在从库上配置主从过程
        - start slave：开启主从复制
        - stop slave：关闭主从复制
        - reset slave all：清空从库所有配置信息
    - 主从复制故障处理
        - 主键冲突，错误代码1062
            - 可以使用percona-toolkit里的pt-slave-restart在从库中跳过错误
        - 主库更新数据，从库找不到而报错，错误代码1032
            - 故障原因：由于误操作，在从库上执行delete删除操作，导致主库上数据不一致
            - 解决方法：
                - 根据报错信息得到binlog文件和position号
                - 在主库上执行mysqlbinlog命令，找到主库上执行的哪条SQL语句导致主从报错
                - 接下来把丛刻丢失的这条数据补上，再执行跳过错误，就能恢复正常了
        - 主从server-id不一致，错误代码1593
        - 跨库操作，丢失数据
            - 解决方法：尽量避免使用库复制的过滤规则，可以在从库上使用replicate-do-db或者replicate-ignore-db等参数，binlog一定要设为row格式，使复制更安全

- 半同步复制
  - 主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待**至少一个**从库接收到并**写到relay log**中，从库再通知主库上的等待线程，才返回给客户端
  - 参数
    - rpl_semi_sync_master_timeout：如果主库等待时间超过该参数设定的时间则关闭半同步复制，转为异步复制
    - rpl_semi_sync_master_wait_point：控制半同步复制下主库在返回session事务成功之前的事务提交方式
      - 值after_commit：
        - 主库将每个事务写入binlog，并传递给从库，刷新到中继日志中，同时**主库提交事务**
        - 然后主库等待从库的反馈，只有接受到从库的回复之后，master才将成功的结果返回客户端
      - 值after_sync：（5.7默认）
        - 主库将每个事务写入binlog，并传递给从库，刷新到中继日志中
        - 然后主库等待从库的反馈，接受到从库的回复之后，master再**提交事务，并将成功的结果返回客户端**
  - 半同步复制需要安装半同步复制插件
  - 操作过程：
    - 基于异步复制，再主从库上分别安装半同步复制插件，并开启复制参数
    - 命令：主从分别配置(show plugins 查看是否加载成功)
      - install plugin rpl_semi_master/slave soname='semisync_master/slave. so' ;
      - set global rpl_semi_sync_master/slave_enable=on;
    - 然后重启slave的IO线程：stop slave io_thread; start slave io_thread
  - 半同步复制和异步复制模式的切换
    - 关闭从库io thread，然后使执行一条SQL语句的时间超过rpl_semi_sync_master_timeout
    - 重新开启从库io thread才又切换回半同步复制
- GTID（全局事务ID）复制：针对事务
  - GTID是个已提交事务的编号，并且唯一
  - 原理
    - GTID由server_uuid和事务id组成，即GTID=server_uuid:transaction_id
    - uuid存放在数据目录的auto.cnf文件下
    - server_uuid是数据库启动时自动生成，每台机器不一样
    - transaction_id是事务提交时由系统顺序分配的不会重复的序列号
  - 存在的价值
    - GTID使用master_auto_posion=1代替了基于binlog和position号的主从复制搭建方式，更便于主从复制搭建
    - 可以知道事务最开始是在哪个实例上提交的

- 方便实现主从之间的failover，再也不用不断地去找binlog和position

- 主从复制中GTID的管理和维护
  - 注意事项
  - 搭建时是在从库中设置：change master to ___，master_auto_position=1；这让操作更加方便可靠
  - gtid_mode参数（只能按修改顺序修改，不可跳跃式修改）
    - OFF：不支持GTID的事务
    - OFF_PERMISSIVE：新的事务是匿名的。同时允许复制的事务可以是GTID，也可以是匿名的
    - ON_PERMISSIVE：新的事务使用GTID。同时允许复制的事务可以是GTID，也可以是匿名的
    - ON：支持GTID的事务

- 与传统复制的切换
  - GTID模式切换到传统模式
    - 先在从库中执行stop slave停掉主从复制，然后调整为传统复制方式，让master_auto_position=0
    - 主从服务器同时调整GTID模式为on_permissive
    - 主从服务器同时调整GTID模式为off_permissive
    - 主从服务器同时关闭GTID模式：enforce_gtid_consistency=off；set global gtid_mode=off
    - 然后把gtid_mode=off和enforce_gtid_consistency=off写入my.cnf文件，下次重启直接生效
    - 测试模式是否切换成功，插入数据，show slave status；GTID没有增加就证明切换成功
  - 传统模式切换到GTID模式
    - 主从库上同时修改参数enforce_gtid_consistency=warn，确保error log中不会出现警告信息，如果有，先修复，再往后执行
    - 主从库上修改参数enforce_gtid_consistency=on，保证GTID一致性
    - 主从服务器同时调整GTID模式为off_permissive
    - 主从服务器同时调整GTID模式为on_permissive
    - 确认从库的ongoing_anonymous_transaction_count参数是否为0，如果为0说明没有等待的事务，就可以执行下一步了
    - 主从服务器同时开启GTID模式：set global gtid_mode=on
    - 查看GTID参数设置是否全开启：show variables like '%gtid%';
    - 先把所有传统复制停掉：stop slave；然后再执行change master to ___ , master_auto_position=1；最后再开启主从复制：start slave;
    - 验证是否切换成功，插入一条数据，show slave status；GTID增加就证明切换成功

- GTID使用的限制条件
  - 不能使用create table ___ select * from ___
  - 在一个事务中既包含事务表的操作又包含非事务表
  - 不支持create/drop temporary table操作

        - 使用GTID复制从库跳过错误时，不支持sql_slave_skip_counter参数的语法
- 多源复制（多个主库数据同步到一个从库里）
    - 搭建过程支持GTID模式和binlog+position方式
    - 主库配置注意
        - 多个主库之间不能有相同数据库名，否则会出现数据覆盖的情况
        - 每个主库上使用不同的复制账号
    - 注意从库的参数配置：复制信息需记录到表中
        - master_info_repository=table
        - relay_log_info_repository=table
    - 从库为每个主库都配置同步过程（使用 for channer ）
- 主从延迟的解决方案及并行复制
    - 从库通过单SQL thread完成任务是出现主从延迟的最核心原因
    - pt-heartbeat来检测延迟的时间大小
        - 在主库先建立一张**heartbeat表**，表中有个时间戳字段
        - 主库上pt-heart的update线程会在指定时间更新时间戳
        - 从库上的pt-heart的monitor线程会检查复制的心跳记录（主库修改的时间戳）
        - 然后和当前系统时间进行对比，得出差异值（延迟的时间大小）
        - 由于heartbeat表中有server-id字段，在监控某个从库的延迟指定参考主库的server-id即可
    - 其他延迟原因
        - MySQL主从同步不是实时同步的而是异步的同步，即主库提交事务以后，从库才再执行一遍
        - 在主库上对没有索引大表的列进行delete或update操作
        - 从库硬件没有主库的好，经常忽略从库的重要性
        - 网络抖动导致I/O线程复制延迟
    - 针对延迟的解决方法
        - 使用5.7版本的并行复制功能（基于组提交的并行复制）
            - 主库并行执行SQL语句，从库也可以通过多个workers线程并发执行relay log中主库提交的事务
            - 可以在从库设置参数slave_parallel_workers > 0
            - slave_parallel_type设置为LOGICAL_CLOCK
        - 可以采用PXC架构实现多节点写入，达到实时同步
        - 业务初期就要选择合适的分表分库策略。避免单表或单库过大，带来额外的复制压力，从而导致主从延迟
        - 其他
            - 避免一些无用的I/O消耗
            - 阵列级别选择RAID10，raid cache策略要使用WB（write back），坚决不要用WT（write through）
            - 系统IO调度要选择deadline模式
            - 适当调整buffer pool大小

        - 避免让数据库进行各种大量运算，可以交给应用端完成或者使用缓存
    - 主从复制的数据校验
        - 如果主库宕机，要进行主从切换，那就必须确保主从数据库数据一致
        - 可以使用percona-toolkit里的pt-table-checksum命令在主库执行**校验查询**或者**检查差异**，然后复制的一致性进行检查，对比主从之间的校验值，并输出对比结果
        - **pt-table-checksum**执行（对主从表中的数据分别进行hash函数运算）完毕，会在主从库里分别生成一张**checksums**表，通过查询该表获得相关信息来判断主从数据是否一致
        - 如果数据不一致就可以使用**pt-table-sync**命令来修复主从不一致的位置，它利用了checksums表的信息
- 数据库优化
    - 配置参数优化
        - innodb_buffer_pool_size
            - 如果但实例的绝大多数都是Innodb引擎表，可以设置为物理内存的50%~80%左右
        - innodb_flush_log_at_trx_commit和sync_binlog
            - redo log和binlog的刷新参数
            - 如果要求数据不能丢失，建议将两值设置为1，可以保证主从架构中数据的一致性，但要保证其强一致性，建议使用5.7的增强半同步功能
        - innodb_max_dirty_pages_pct
            - 脏页占innodb buffer pool的比例，当比例达到设定值时，触发刷新脏页到磁盘
            - 不要将该值设置太大，脏页过多也会影响数据库的TPS，建议调整为25~50（%）
        - innodb_io_capacity
            - innodb后台进程最大IO性能指标，影响刷新脏页和插入缓冲的数量。默认值是200
            - 在高转速磁盘下，应适当调高该参数
            - SSD磁盘配置下可以调整该值为5000~20000，PCIE-SSD可以调整得更高（50000左右）
        - innnofb_data_file_path
            - 建议设置为ibdata1:1G:autoextend
            - 该参数不要使用10M，一般设置为1G，防止在高并发情况下，数据库受到影响
        - long_query_time
            - 建议设置该值为0.1~0.5，记录那些执行较慢的SQL
        - binlog_format
            - 建议binlog格式设为row模式，数据更安全可靠，复制过程不会出现丢失数据的情况
        - interactive_timeout_，wait_timeout
            - 交互/非交互时等待时间，两个参数最好一致，且必须同时修改
            - 建议调整为300~500，不要默认8小时

        - max_connections
            - 数据库最大连接数，谨慎调高该值
            - 在调高此参数的同时，还应该调低interactive_timeout_，wait_timeout的值
        - innodb_log_file_size
            - redo log值太大，实例恢复时会占用大量时间
            - redo log值太小，会造成日志切换过于频繁
        - gerenal log
            - 全量日志建议关闭，否则会导致磁盘空间紧张
    - 表设计及其他优化
        - 金钱、日期时间、IPv4地址尽量使用int类型来存储，日期也可以选用datetime它只比timstamp大1字节
        - text、blod这种大字段不建议与业务表放一起
        - SQL语句中尽量避免or语句，这种判断语句可以交给程序自行完成，不要交给数据库，也要避免union，尽量使用union all，减少去重和排序工作
        - select查询表时只获取必要字段，可以减少网络带宽，还可以利用覆盖索引，选择性低的字段不要建立索引
        - 单表索引不要超过4~5个，当执行DML语句时也会对索引进行更新，如果索引太多会造成索引树的分裂，性能也会下降
        - 模糊查询'%___%'不要出现在数据库中，可以用搜索引擎sphinx代替
        - 索引字段不要使用函数，否则用不到索引
        - 执行计划extra项看到Using filesort或Using temporary时，也要优先考虑在排序列和分组列上建立索引
        - limit子句上的优化，建议使用主键来进行范围检索，缩小结果集大小，是查询更高效
        - 可以使用Mysql的show profile来查看语句执行资源消耗情况，要开启profiling，使用show profiles; / show profile block io, cpu for query 1;
        - 使用show global status命令来查看数据库运行状态，通过得出数值，优化Mysql运行效率
- Druid
- Redis
- ElasticSearch