Mysql 原理

- 一条sql查询语句怎么执行的
  - 基本架构
    - 客户端：service层包括核心服务功能，以及所有内置函数
      - 连接器：管理连接，权限验证
      - 查询缓存：加快查询效率，但是表更新缓存不在，8以后版本取消缓存
      - 分析器：词法语法分析，判断列名，表名是否与真实表对应
      - 优化器：执行计划生成，索引选择
      - 执行器：操作引擎，返回结果
    - 存储引擎
      - 数据储存，提供读写接口、储存过程，触发器，视图
      - 架构：插件式，支持InnoDB,MyISAM,Memory储存引擎，Mysql 5.5.5 开始使用InnoDB为默认存储引擎
- 一条sql更新语句是如何执行的
  - redo log(重做日志),引擎日志
    -
      - InnoDB先把记录写到redo log 中，并更新内存，系统空闲时，引擎将记录更新到磁盘中
      - 数据块发生重启，之前提交的记录都不会丢失，称为crash-safe
  - binlog(归档日志)，server 层日志
    - binlog 日志为逻辑记录，记录执行了些什么原始操作，而redolog 记录的是数据页改变了什么
    - binlog 可以追加写入，而redolog 物理日志，空间会用完，所以需要清除数据，同步到磁盘才能写入
  - 执行更新操作步骤（两阶段提交，保证两个状态逻辑上一致）
    - 执行器找引擎取对应行，如果在内存中就直接返回给执行器，不然就从磁盘读入
    - 执行器拿到数据，更新，调用引擎接口写入新数据
    - 引擎将新数据更新到内存，将更新操作记录到redolog ， redolog处于prepare 状态
    - 执行器生产这个操作的binlog，把binlog 写到磁盘
    - 执行器调用引擎提交事务接口，将redo log 改成commit 状态
- 事务隔离;为什么我改了还看不到
  - 事务特性
    - ACID(Atomicity、Condidtrncy、Isolation、Durability),原子性、一致性、隔离性、持久性
  - 事务出现问题
    - 脏读、不可重复读、幻读等问题
  - 解决事务问题，需要事务隔离
    - 读未提交、读已提交、可重复读、串行化

- 可重复读：在事务执行过程中看到的数据，总是跟启动时看到的数据一致，意思是其他事务修改对他没影响
- 串行化：对于同一行记录，写、读都会加锁，出现读写冲突，必须等到前一个事务完成才执行

- 怎么实现事务隔离
  - 通过视图解决
    - 可重复读视图在事务启动时创建
    - 读已提交视图在每个sql 执行的时候创建
    - 读未提交直接返回记录最新值，没有视图概念
  - 可重复读适用场景
    - 比如银行校对明细表时，不想在核对时候，新增明细干扰核对结果
  - 回滚原理
    - 记录更新时会记录一条回滚操作
    - 不同时刻启动事务有不同的read-view ， 同一记录在系统中可以存在多个版本，就是数据库多版本并发控制
    - 在系统不需要回滚日志时删除，就是在系统没有比这个回滚日志更早的read-view时
  - 事务启动方式
    - 显示启动 begin 或者 start transaction，配套语句commit,回滚语句是rollback
    - set autocommit=0，将线程自动提交关闭，执行语句事务就启动，等到手动commit或者 rollback 时或断开连接时关闭事务
  - 查看数据库中存在长事务
    - select * from information_schema.innodb_trx where TIME_TO_SEC(timediff(now(),trx_started))>60

- 索引
  - 常见的索引数据
    - hash表、有序树、搜索树
      - hash适用于等值查找，不适合范围查找
      - 有序数组在等值查询和范围查询中性能非常优秀，但是只适用于静态存储引擎
      - 二叉搜索树为了保证查询复杂度为Ologn 需要维护为平衡二叉树
      - 在数据库中，为了减少树的高度（索引需要存储到磁盘，每次读取需要访问磁盘）使用N叉树
    - InnoDB 索引模型
      - 使用的是B+树，索引分为主键索引（聚簇索引）和非主键索引（二级索引）
      - InnoDB是索引组织表，一般使用自增主键，这样不会因为插入重排，提高效率，主键长度越小，普通索引的叶子节点越小，占用空间越小
    - 基于主键索引和普通索引查询区别
      - 只需要搜索一颗B+树
      - 先搜索K索引树、得到ID,通过ID再搜索D树
    - 覆盖索引：可以减少树搜索次数，显著提升查询性能
    - 最左前缀原则
      - 在模糊查询 name like 张%
      - 建立联合索引，考虑如何安排索引内的字段顺序

- 索引下推：在索引遍历过程中，对索引中包含的字段做判断，直接过滤掉不玩朱条件的记录
- 原则：尽量少访问资源是数据库设计的重要原则之一，在使用数据库时，设计表时，尽量减少资源消耗为目标

- 全局锁和表锁
  - 基本信息
    - 设计初衷——解决并发问题
    - 锁分成三类：全局锁、表级锁、行锁
  - 全局锁
    - 对整个数据库进行加锁
      - mysql 全局读锁方法：FTWRL(Flush tables with read lock)
        - 针对于像MyLSAM不支持事务引擎
      - 其他线程的DML、DDL、更新事务提交语句都会被阻塞
    - 使用场景：全库备份
    - 缺陷
      - 全库只读，备份期间不能执行更新，业务基本停摆
      - 如果从从库备份，不能执行主库同步的binlog,导致主从不一致
    - 使用可重复读级别开启事务
      - Mysql备份工具：mysqldump，使用single-transaction，在备份是会启动事务，确保拿到一致性视图
    - 不使用set global readonly=true 方式原因
      - readonly 会被用来做其他逻辑，判断库是否是主库，修改global 变量方式影响大，在从库的super权限readonly无效
      - 异常处理机制异常,FTWRL客户端断开 Mysql自动释放全局锁，而readonly 不会
  - 表级锁
    - mysql两种表级锁：表锁、元数据锁
      - 表锁
        - lock tables ...read/write
        - 可使用unlock tables 释放锁
        - InnoDb可支持行锁，一般不使用lock tables 控制并发
      - MDL (metadata lock) 元数据锁
        - 在mysql 5.5 加入
          - 在增删查改加入读锁
          - 当对表结构变更加入MD写锁
        - 读锁不互斥、写锁互斥、读写锁互斥
  - 如何安全地给小表加字段?
    - 解决长事务，事务不提交会一直占用MDL锁，考虑先暂停DDL，或kill长事务
    - 在热点表中，kill事务未必有用，就在alter table 语句中设定等待时间，在指定时间内拿不到MDL先放弃。（NOWAIT/WAIT）
  - MDL 会直到事务提交才释放，在做表结构变更的时候，你一定要小心不要导致锁住线上查询和更新。
- 行锁

- 两段锁协议
  - 在 InnoDB 事务中，行锁是在需要的时候才加上的，但并不是不需要了就立刻释放，而是要等到事务结束时才释放。这个就是两阶段锁协议。
  - 如果你的事务中需要锁多个行，要把最可能造成锁冲突、最可能影响并发度的锁尽量往后放。
- 死锁和死锁检测
  - 出现循环资源依赖出现死锁
  - 解决死锁方法
    - 直接进入等待，直到超时
      - 通过：innodb_lock_wait_timeout 设置 默认50s
    - 发起死锁检测
      - 发生死锁后，主动回滚死锁链条的某一事务，让其他事物得以执行
        - innodb_deadlock_detect 这种为on
    - 死锁检测消耗：每个被阻塞的线程都会检测是否因为自己加入导致死锁，复杂度O(n),1000个线程死锁检测100万量级，消耗大量cpu
  - 解决死锁检测cpu消耗问题
    - 关闭死锁检测，死锁后通过业务重试，业务无损；但是出现大量超时，业务有损
    - 控制并发度
      - 控制并发量，限制并发数量
        - 修改mysql源码，对于相同行的更新，在进入引擎前排队
        - 考虑将一行的数据改成逻辑上的多行减少锁冲突

- 事务是否是隔离的
  - 事务的启动时机
    - begin/start transaction 命令并不是一个事务的起点，在执行到它们之后的第一个操作 InnoDB 表的语句，事务才真正启动。如果你想要马上启动一个事务，可以使用 start transaction with consistent snapshot 这个命令。
    - 启动方式
      - 第一种启动方式，一致性视图是在执行第一个快照读语句时创建的；
      -
第二种启动方式，一致性视图是在执行 start transaction with consistent snapshot 时创建的。
  - 两个视图
    - view视图
      - 它是一个用查询语句定义的虚拟表，在调用的时候执行查询语句并生成结果。创建视图的语法是 create view … ，而它的查询方法与表一样。
    - 一致性视图
      - InnoDB 在实现 MVCC 时用到的一致性读视图，即 consistent read view，用于支持 RC（Read Committed，读提交）和 RR（Repeatable Read，可重复读）隔离级别的实现。
  - 快照
    - 实现一致性视图
      - InnoDB 利用了“所有数据都有多个版本”的这个特性，实现了“秒级创建快照”的能力。
  - 事务视图可见性

- 版本未提交，不可见
- 版本已提交、但是在视图创建后提交，不可见
- 版本已提交，但是视图在创建前提交，可见

- 更新逻辑
  - 更新数据都是先读后写的，而这个读，只能读当前的值，称为“当前读”（current read）。
  - 当前读
    - 事务中更新操作时候，尽管有事务一致性视图，但是保证在事务未提交时候，别的事务已经修改数据丢失
    - 在select 加锁 加lock in share mode 或者 for update 的时候也是 当前读，一致性视图可见性失效
- 两段锁协议
  - 事务c' 未提交，更新数据时写锁占用，当前事务B是当前读，读最新版本的数据，这时就会等待c'锁释放
- 可重复读怎么实现的?
  - 核心是一致性读，但是事务更新操作的时候是当前读，如果当前读的记录行锁被占用就进入锁等待
- 读提交和可重复读的区别
  - 可重复读:
    - 在事务开始创建一致性视图，之后查询都用这个视图
    - 在读提交隔离级别下，每个语句执行前都会重新计算新的视图

- 选择普通索引和唯一索引
  - 对查询语句影响
    - 唯一索引在查到满足条件时会停止检索，而普通索引会进行判断
    - 普通索引检索数据在本页最后一条时会检索下一页
  - 对更新语句影响
    - 普通索引
      - 使用change buffer
    - 唯一索引
      - 每次更新判断唯一索引是否冲突
    - 记录更新在内存中
      - 唯一索引
        - 判断冲突，插入数据
      - 普通索引
        - 插入值
    - 记录更新不在内存中
      - 唯一索引
        - 将数据读入内存、判断是否冲突、插入值
      - 普通索引
        - 更新记录在change buffer 中，语句执行结束
    - 数据库成本最大操作时读入内存涉及的随机IO访问，大量插入会导致数据库

- change buffer
  - change buffer 的使用场景
    - 只限于用在普通索引中，不适合用于唯一索引
    - 适用于读多写少业务，并且在写完后马上被访问的概率小，changebuffer使用效果更好，比如账单类、日志类系统
  - change buffer 和redo log 区别
    - redo log 主要节省的是随机写磁盘的 IO 消耗（转成顺序写），而 change buffer 主要节省的则是随机读磁盘的 IO 消耗。
  - change buffer merge 过程
    - 从磁盘读入数据页到内存
    - 重changge buffer 里找出change buffer 记录，依次应用到数据页，生成新数据页
    - 写redo log。这个redo log 包含数据变更和change buffer 变更
- 两个索引区别
  - 在查询上区别不大，主要是在更新时对性能影响大

- 为什么Mysql 有时会选错索引
  - 优化器逻辑
    - 找到最优执行方案，用最小的代价执行语句，扫描行是影响执行代价的因素之一、除此之外还有是否使用临时表、是否排序
    - 扫描行怎么判断
      - 根据索引的基数判断，基数越大，区分度越好
        - 通过采样统计，选择N个数据页，统计页面上不同的值，得到平均值，然后乘以索引页数
        - 统计信息有两种存储方式：持久化存储、存储在内存中
    - 可以使用 force index(a) 进行强制设置索引，但是优化器会对索引进行权衡，会把回表操作的时间算进入，所以扫描行数并不是唯一影响优化器判断的条件，使用analyze table t 重新统计索引
  - 索引选择异常和处理
    - force index 强行选择一个索引
    - 通过sql 引导优化器选择正确索引
    - 在某些场景下，可以新建更适合的索引，提供优化器做选择，或者删除误用索引

- mysql 怎么保证主备一致
  - 主要通过binlog日志保证主从一致，高可用架构都依赖binlog
  - 主备一直原理
    - Mysql 主备切换，实际生产使用双M流程
      - 主备流程
  - binlog 三种格式
    - statement
      - 在 limit 的时候，主备库使用索引不一致的情况下，就会导致操作数据不一致
    - row
      - 占用空间大，删除10万行就是要写十万行日志
      - 优点：回复数据

- 在更新数据时会保存之前数据，只要交换数据位置就能恢复数据
- mixed
  - mixed 存在场景：statement 格式可能导致主备不一致，但是row 占用空间大，就采用折中办法
- 解决循环复制问题
  - 双M结构
    - 在A更新一条语句，生产的binlog 发给节点B,B也生成binlog，同时A也是B的备库，怎么防止循环更新
    - 通过serverId 不同来区分，备库生成的binlog 的serverID 与主库一致，主库执行binlog，发现serverID一致就不执行
- 怎么给字符串加索引
  - 在字符串建立索引时支持前缀索引
    - 使用前缀索引，定义好长度，就可以做到既节省空间，又不用额外增加太多的查询成本
    - 建立前缀索引时候，在查询的时候，在前缀索引重复时会回表根据主键索引查询，降低查询效率
  - 前缀索引对覆盖索引的影响
    - 使用前缀索引用不上覆盖索引的优化，就算前缀索引的字符全部包括，也需要回表根据ID进行查询
  - 其他方式
    - 使用倒叙存储
    - 使用hash字段
- 数据库有时候为什么会抖一下
  - 数据库在同步relog 到磁盘（刷脏页）过程中查询会变慢
  - 引发数据库flush过程几种场景
    - 1、InnoDB 的redo log 写满，停止所有更新操作，把checkpoint 往前推进
    - 2、系统内存不足，淘汰脏页的时候flash数据到磁盘
    - 3、系统空闲时刷脏页
    - 4、在mysql正常关闭，把脏页flush到磁盘，在数据库启动时，直接从数据库读，速度更快
  - 内存管理
    - InnoDB使用缓冲池（buffer pool）管理内存
      - 缓冲池的内存中有三种状态：未使用、使用并且干净、使用并且脏数据
      - 读入数据到内存必须到缓冲区申请数据页：使用最久未使用淘汰算法，淘汰的如果是脏页就需要刷新到磁盘
        - 淘汰脏页多，查询时间变长
        - 日志写满，不能更新
  - InnoDB刷盘速度
    - 通过innodb_io_capacity 设置主机IO能力
    - 刷新速度：脏页比例、redo log 写盘速度
    - 合理设置InnoDB_io_capacity & 保证脏页比例不接近75%
  - 利用WAL （write ahead logging） 预写日志，利用redolog把随机写转换为顺序写，提升数据库IO性能
- 表数据删掉一半，表文件大小不变

- InnoDB 包含两部分:表结构定义、数据
  - mysql 8.0 前表定义放在.frm 后缀文件中，8.0后可放在数据表中，空间占用小
    - innodb_file_pre_table　5.6.6 默认NO, 代表单独存放.ibd文件 ， OFF 存放在系统工共享表空间，数据字典放一起
    - 建议设为ON，单独存储容易管理，不需要表设置drop table 直接可删除文件，空间可回收
  - 删除数据只是标识数据已删除，能复用，空间未释放
    - 单条数据删除
    - 数据页被删除
    - 表数据删除
    - 插入数据导致数据页分裂
- 解决方法
  - 重建表
    - alter table A engine=InnoDB
      - 数据转存、交换表明、删除旧表 5.6前版本
      - 5.6 后引入Online DDL
        - 建立临时表、扫描表A主键数据页
        - A中记录生成B+树，存储到临时表中
        - 生成临时文件时，更新操作记录在row log 中
        - 临时表生成后，将日志文件应用到临时文件
        - 临时文件替换成A
      - 在alert 执行时获取DML写锁，在copy data 时退化成读锁
        - 为了实现Online MDL 读锁不会阻塞增删改
        - 不直接用读锁，禁止其他线程对表同时做DDL
      - online 与 inplace 区别
        - DDL过程如果是Online ,就一定是inplace
        - inplace是DDL，可能不是Online
          - Mysql 8.0 添加全文索引（FULLTEXT index）和 空间索引（SPATIAL index）
- 三种重建方式区别
  - optimize table
    - recreate+analyze
  - analyze table
    - 只是对表索引做统计、未修改数据
  - alter table
    - recreate
- count(*) 为什么越来越慢
  - 实现方式
    - MYISAM 引擎 由于不支持事务，直接把count 保存到磁盘中
    - InnoDB 执行count(*) 需要从引擎中读取
  - InnoDB为什么需要每次扫描行数
    - 因为支持事务操作，支持MVCC，InnoDB返回行数不确定

- count(*)优化
  - mysql原则：保证数据前提下，尽量减少扫描的数据量，是数据库统计通用法则
  - 普通索引树节点是主键值，只需要找最小的普通索引树查找
- 解决思路
  - 将计数保存在缓存系统中的方式，还不只是丢失更新的问题。即使 Redis 正常工作，这个值还是逻辑上不精确的。
  - 单独用一张表存储，在更新时加事务
- 不同的count性能
  - count(字段)<count(主键 id)<count(1)≈count(*)，所以我建议你，尽量使用 count(*)。
- order by 工作原理
  - 全字段排序
    - 在sort_buffer 中快速排序，设置sort_buffer_size
    - 在磁盘中归并排序
      - 需要额外的临时文件
  - rowid 排序
    - mysql认为排序单行长度太大时就只会把排序字段和ID放入内存
  - 思想：如果内存够就使用内存，减少磁盘访问
  - 并不是所有都需要排序
    - 使用组合索引，能达到覆盖索引目的
- 如何正确显示随机消息
  - 场景：在表插入10000行数据记录、随机选择3个记录
    - 方法一：order by rand()
      - 需要使用临时表、执行排序操作，查询代价大
      - 内存表，回表只是简单地根据数据行位置，直接访问内存得到数据，排序行越小越好，mysql选择rowId 排序
      - 当tmp_table_size 内存临时表大装满，就会使用磁盘临时表
        - 根据数据是否超出sort_buffer_size 判断使用优先队列排序算法还是归并算法
    - 使用最多方法
      - 取得整个表的行数，并记为 C。
      - 取得 Y = floor(C * rand())。 floor 函数在这里的作用，就是取整数部分。
      - 再用 limit Y,1 取得一行。
  - 扩展：rowID
    - 对于有主键的 InnoDB 表来说，这个 rowid 就是主键 ID。
    - 对于没有主键的 InnoDB 表来说，这个 rowid 就是由系统生成的。
    - MEMORY 引擎不是索引组织表。在这个例子里面，你可以认为它就是一个数组。因此，这个 rowid 其实就是数组的下标。
- sql 优化
  - 条件字段操作
    - 对索引字段做函数操作，可能会破坏索引值的有序性，因此优化器就决定放弃走树搜索功能。
    - 加了函数操作，Mysql无法使用索引快速定位功能，只能用全索引扫描

- 隐式类型转换
  - 参数隐式转换：字符与数字比较会转换成数字进行比较，相当于执行内置函数,判断主键索引和普通索引大小进行选择索引,但是全表扫描
- 隐式字符编码转换
  - 字符集 utf8mb4 是 utf8 的超集，所以当这两个类型的字符串在做比较的时候，MySQL 内部的操作是，先把 utf8 字符串转成 utf8mb4 字符集，再做比较。
  - 连接过程中要求在被驱动表的索引字段上加函数操作，是直接导致对被驱动表做全表扫描的原因。
- 解决思路
  - 不让索引字段做函数转换操作，在条件参数上做转换，或者拆分sql

- 查询一行，为什么速度这么慢
  - 表被锁住，执行show processlist 查看语句处于什么状态
  - 查询时长不返回，lock timeout
    - 1、等待MDL锁，现在有一个线程正在表 t 上请求或者持有 MDL 写锁，把 select 语句堵住了。
    - 2、等flush（表关闭）
    - 3、等行锁
  - 查询慢
    - 查询慢，select 加锁反而更快，因为lock in share mode 是当前读，而查询id=1,一致性读，需要回滚
  - 扫描行数多：坏查询不一定是慢查询
- 幻读是什么?
  - 指的是一个事务在前后两次查询范围内，后一次看到了前一次没看到的数据，当前读不是幻读，幻读是'指新插入的行'
  - 存在问题
    - 语义上不一致
    - 数据不一致，数据和binlog日志
  - 解决幻读
    - 引入间隙锁，锁的是两个值之间的间隙
      - 间隙锁冲突只是：往间隙插入记录
      - 间隙锁和行锁合称：next-key lock
  - 带来的问题
    - 间隙锁的引入，可能会导致同样的语句锁住更大的范围，这其实是影响了并发度的
- next-key lock 间隙锁与行锁
  - 加锁规则
    - 1、加锁的基本单位next-key lock,前开后闭区间
    - 2、查找过程中访问到对象才加锁
    - 3、索引等值查询、唯一索引加锁，next-key lock 退化成行锁
    - 4、索引等值查询，向右遍历且最后一个不满足等值条件，next-key lock 退化成间隙锁
    - 5、唯一索引范围查询访问到不满足条件第一个值为止
  - 锁案例

- 等值查询间隙锁
- 非唯一索引等值锁
- 主键索引范围锁
- 非唯一索引范围锁
- 唯一索引范围锁bug
- 非唯一索引存在‘等值’
- limit 语句加锁
- 子主题

- 结论
  - 锁是加载索引上的
  - lock in share mode 查询使用覆盖索引
  - for update ， 系统认为需要更新数据，会给主键索引加行锁
  - 在delete limit 加锁后满足limit值后循环结束

- 紧急提升mysql性能
  - 短连接风暴
    - 链接数据库执行很少sql，就断开，下次需要重连
    - 解决方式
      - 直接修改max_connections 值
        - 可能会导致系统负载过大，大量消耗在权限验证等逻辑，链接拿到CPU 资源执行业务SQL
      - 处理占着连接不工作线程
        - 通过kill connection +id 主动踢掉，或者设置wait_timeout
        - 如果是连接数过多，你可以优先断开事务外空闲太久的连接；如果这样还不够，再考虑断开事务内空闲太久的连接。
      - 减少连接过程中的消耗
        - 让数据库跳过权限验证阶段
          - 重启数据库，并使用–skip-grant-tables 参数启动。这样，整个 MySQL 会跳过所有的权限验证阶段，包括连接过程和语句执行过程在内。
  - 慢查询性能
    - 索引没设计好
      - 创建索引支持Online DDL，可alter table
    - SQL 语句没写好
      - 在数据库执行查询重写
    - Mysql 选错索引
      - 给语句加上force index
    - 解决方式
      - 把慢查询日志打开，并把long_query_time 设置成0，确保语句记录入慢查询日志
      - 模拟线上数据，做回归测试
      - 观察慢查询日志输出
  - QBS

- 程序bug 导致某语句QPS 暴增，也可能导致mysql 压力过大
- 规范运维体系：虚拟化、白名单、业务账号分离

- mysql 怎么保证数据不丢失
  - binlog 写入机制
    - 事务执行过程中、把日志写到binlog cache,事务提交在时，再把binlog cache 写到binlog 文件中
    - 原则：无论事务多大，事务binlog 不能被拆开，确保一次性写入
    - 设置sync_binlog 100—1000提升性能，在积累N个事务提交后才fsync
  - redo log 流程
    - 1、存在redo log buffer 2、写到磁盘，存放在文件系统page cache 3、持久化到磁盘
    - 不需要每次写入磁盘、事务没提交，redo log buffer 中的日志可能被持久化到磁盘
      - 后台线程每秒轮询一次
      - redo log buffer 占用空间达到InnoDB_log_buffer_size 一半，后台线程主动写盘
      - 并行事务，事务B 设置 innodb_flush_log_at_trx_commit=1，redo_log_buffer 全部持久化到磁盘，为提交事务A redo_log一并提交
      - 每秒一次后台轮询刷盘，再加上崩溃恢复这个逻辑，InnoDB 就认为 redo log 在 commit 的时候就不需要 fsync 了，只会 write 到文件系统的 page cache 中就够了。
    - 优化
      - 组提交
        - 一次组提交里面，组员越多，节约磁盘 IOPS 的效果越好
      - 延时提交
        - 第一个事务写完 redo log buffer 以后，接下来这个 fsync 越晚调用，组员可能越多，节约 IOPS 的效果就越好
  - WAL(write ahead log)
    - redo log 和binlog 都是顺序写，比随机写速度快
    - 组提交机制，大幅度降低磁盘的IOPS消耗