# Mysql事务日志

原子性：undolog实现

持久性：redo log实现

隔离性：基于锁机制、数据的隐藏列（mvcc）、undo log与类next-key lock机制实现

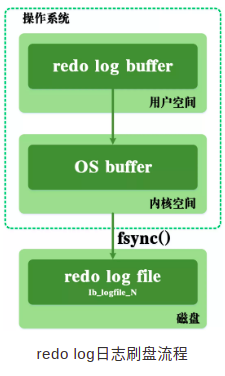
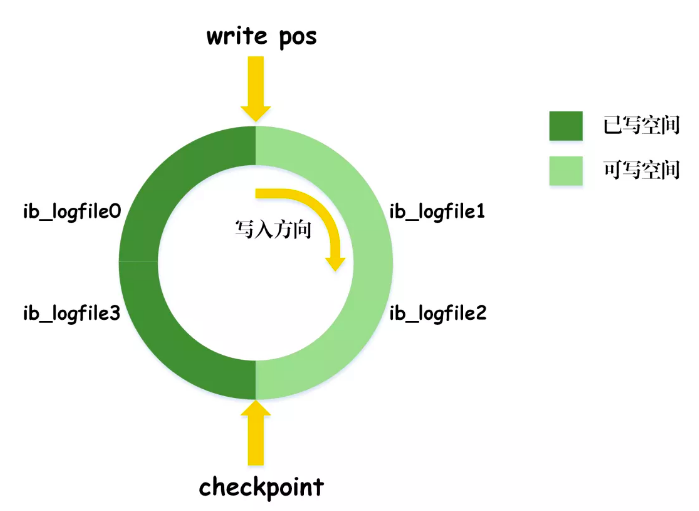
一致性：上面三者皆为此服务

Innodb通过MVCC解决脏读、不可重复读，用next-key lock机制解决幻读

所以 可串行化的隔离级别 是 避免脏读、不可重复读和幻读的充分不必要条件

## Redo log

重做日志，innodb引擎层日志，记录事务操作引起的事务变化，即更新操作，记录数据页的物理修改。

  
 write\_pos：开始写入的位置

Checkpoint：内存满后，开始檫除的位置，即开始刷脏数据的位置

redo日志先存入的是内存中的环形缓冲区，write\_pos和chenkpoint中间的空间可用于写入新数据，当write\_pos追上checkpoint时，表示redo log日志已经写满，就会出发checkpoint规则，将buffer中的脏数据页和脏日志页刷到磁盘。

所以redo日志分为两个部分：

一是存在易失性内存中的缓存日志redo log buff

二是保存在在磁盘中的redo log file

## Binlog

二进制日志是服务层日志，用来记录数据库变化，当然页包含了更新操作。因此用作主从的同步。

但和redo log不同的是，binlog是逻辑日志，记录的是sql语句的原始逻辑，跟undo log有点类似，只不过undo日志记录的是sql语句的反向逻辑。

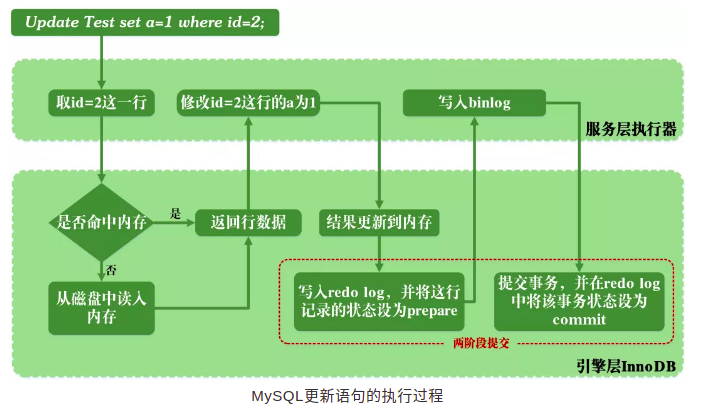
物理日志：数据库中数据页上的变化信息，只看中结果，不在乎何种路径导致该结果

逻辑日志：通过某一方法使数据发生变化，存储的是逻辑性的操作。

Redo log用于mysql宕机后数据回复，binlog可以基于时间点对数据进行恢复和备份。

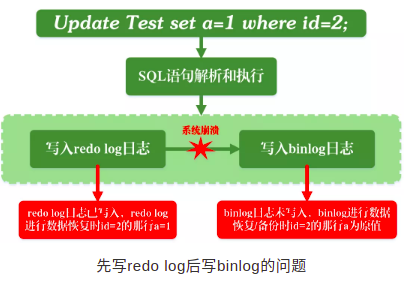
Binlog是服务层日志，因此所有引擎都能用，但是binlog只能提供归档的作用，所以才有了redo log。

Binlog是追加写入，不会覆盖已写文件，而redo log是循环写入和檫除。



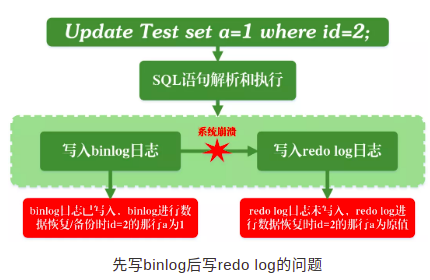
如图是redo log写入的两阶段提交，如果采用单阶段提交，将会导致原先数据库的状态和被恢复后的数据库状态不一样，下面就两种单阶段提交讨论：

1. 先写入redo log，后写入binlog：



即从数据库可能没有同步更新到

1. 先写入binlog，后写入redolog



即根据数据恢复后，恢复的数据相比于从数据库会不一致

综上：就是会导致主从数据库不一致。

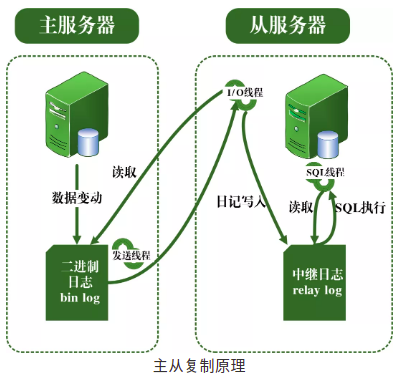
## **Undo log**

逻辑日志，对sql语句执行相关信息记性记录，回滚时，就是执行与sql语句相反的sql语句，把数据改回去。

Undo log有两个作用：

1、提供回滚 2、实现mvcc

## 主从复制



1. 发送线程，发送binlog日志到从服务器
2. IO线程，读取主线程发送过来的binlog日志内容，并拷贝到本地的中继日志
3. SQL线程，读取中继日志中关于数据更新的SQL语句并执行，实现主从一直。

好处：

1. 数据异地备份
2. 读写分离，提高并发
3. 负载均衡

# 索引

## 为啥索引常用 B+ 树作为底层的数据结构

* 完美满足索引的三个条件：
* 1、根据某个值精确快速查找，减少扫描行数
* 2、支持范围查找，将随机IO变为顺序IO
* 3、索引值已经排好序，无需创建临时表，并支持顺序和逆序查找

## 除了 B+ 树索引，你还知道什么索引

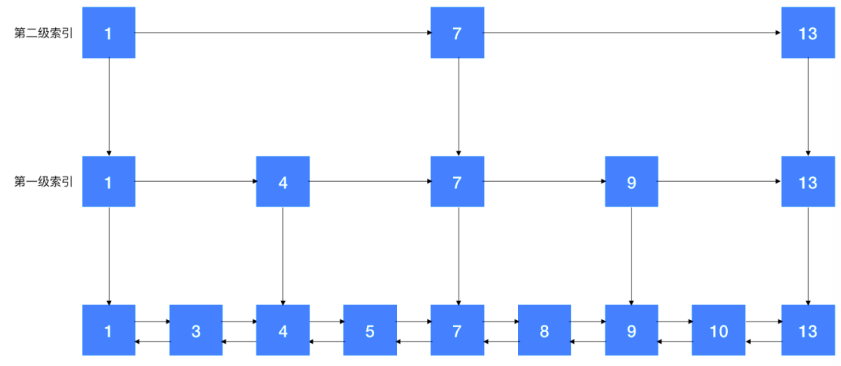
哈希索引：

优势：基于散列表，结构紧凑，查找快速

劣势：只满足精确值匹配，不支持范围查询，那当然也就无法用于排序，存在 大量随机IO和创建临时表操作，

跳表：

多层链表，支持快速查找，范围查找，正序逆序查找，与B+树有异曲同工之妙。



## 为啥推荐自增 id 作为主键，自建主键不行吗

虽然说主键只需满足唯一性即可，但是自键主键是无序的，这样插入数据时，插入主键索引树的某一个节点，那么该节点进行页分裂的可能行就会很大，无疑会消耗性能，但是自增id的话，始终在树的叶子节点插入数据，所以不会存在页分裂

## 什么是页分裂，页合并

页的概念：无论是内存还是硬盘，操作系统都是按页的大小进行读取的，磁盘每次读取都会预读，根据局部性原理，将连续的数据读入内存，那这个连续的内存有多大呢？必须是操作系统页大小4kb的整数倍，mysql的页默认为16kb，所以B+树的节点就是一个mysql页。

这里有一个误区，就是mysql页是不是越大越好，因为这样的话，树就显的更矮胖，IO就越小，这里就涉及到缓冲池了，InnoDB是通过内存中的缓冲池来管理磁盘中读取的页数据的，页太大的话，很快就会把这个缓冲池给撑满，那么页就会在内存和磁盘频繁换入换出，影响性能，所以最好默认为16kb。

页分裂：插入操作导致页的大小超过16kb

页合并：删除操作导致页的大小下雨16kb

## 什么是索引，索引的作用

索引应该尽量满足的三个条件：

1. 快速精确索引，减少扫描行数，哈希、B+树索引都能很好满足这点
2. 支持范围扫描，化随机IO为顺序IO，随机IO的寻道时间是噩梦，哈希索引是无序的，不满足这点。
3. 索引有序，这样就不必创建临时表（内存临时表装不下会生成磁盘临时表）去排序。
4. 综上：熟记避免全盘扫描，减少随机IO，避免排序。

随机IO和顺序IO相比：

1. 多次寻道，寻道时间是非常大的
2. 寻道之后移动到目的扇区的时间同样是噩梦。
3. 基于前两者，彼此时间相差百倍

## 高性能索引策略

几种不常见的导致索引失效的情况：

1. 隐式类型转换：

SELECT \* FROM tradelog WHERE tradeid=110717;

Tradeid类型为varchar

**SELECT** \* **FROM** tradelog **WHERE** **CAST**(tradid **AS** signed int) = 110717;

这样的话，相当于给表达式左边加了函数

1. 隐式编码转换：

**SELECT** d.\* **FROM** tradelog l, trade\_detail d **WHERE** d.tradeid=l.tradeid **AND** l.id=2;

Tradelog和trade\_detail两张表的编码不同

**SELECT** d.\* **FROM** tradelog l, trade\_detail d **WHERE** (**CONVERT**(d.traideid **USING** utf8mb4)))=l.tradeid **AND** l.id=2;

同上

1. 使用order by 上的列索引，但是select的数据却不仅仅是order by上的列

如：**SELECT** \* **FROM** **user** **ORDER** **BY** age **DESC**

**这是因为我们使用select \* ，导致回表，而不能实现索引覆盖，这样的话会导致大量的随机IO，还不如全盘扫描的顺序IO快，所以不会使用索引。**

## 索引设计准则：三星索引

索引的设计一般就是将选择性最高的列防在索引的最前列，但是通常不如避免随机IO和排序重要，这里引入一个准则：

第一颗星：where后面参与查询的列可以组成单列或联合索引---避免全表扫描。

第二颗星：使用order by column时，取出的结果集就是排好序的，也就是select的列要带有order by的列，无需创建临时表操作。---避免排序

第三颗星：select对应的列尽量使索引列，里面回表------避免随机IO

# MyISAM引擎和InnoDB引擎

MyISAM引擎：

不支持事务，当然也就不支持CRUD，这一设计是为了性能和效率考虑的

不支持外键

表级锁，所以并发性差

在磁盘存储三个文件，文件名和表名相同，拓展名为.frm（存储表定义） .MYD（mydata，存储数据）.MYI(MYIndex，存储索引）. 注意MyISAM只存储索引文件，不缓存数据文件。

索引：全局索引、B树索引、R-Tree索引

不存在事务日志，所以数据文件损坏，难以恢复。

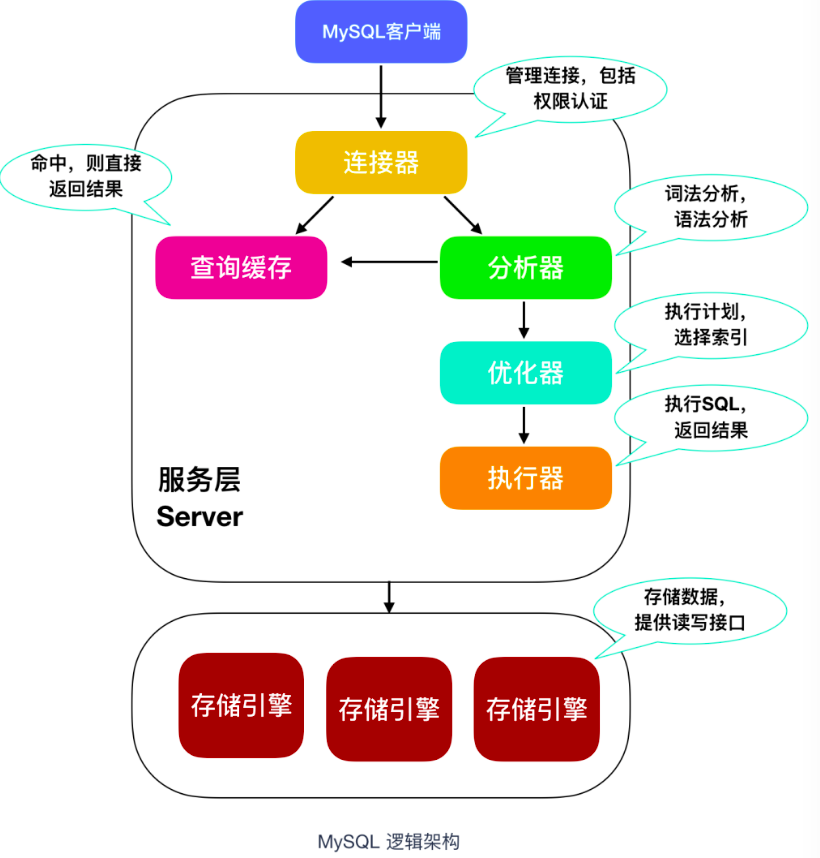
InnoDB引擎

也有.frm文件存储表结构定义，但是不同的是，InnoDB的表数据与索引数据存储在一起，后缀名为.ibd，位于B+树的叶子节点上。

行锁的开销更大，容易发生死锁。

查询性能： MyISAM优于Innodb，因为前者是B树结构，直接定位到数据所在内存地址，而无需多次定位数据块。但是增删改的话还是Innodb更好，因为B树会有大量的页分裂和页合并操作。

# Mysql逻辑架构



Mysql大致分为Server层和存储引擎层，如上：

Kundb基于的vitess开发，主要是服务层的开发，带有分析器（词法语法解析），优化器（生成执行计划，但是不分逻辑计划和物理计划）,执行器（Execute函数）。

Mysql所有跨引擎的的功能在服务层实现，而kundb所有跨库的操作当然也在服务层实现。

## 查询缓存

查询缓存不建议使用，因为只要对某一张表执行了更新操作，那么所以查询缓存就会失效，UI与更新频繁的数据库来说，查询缓存的命中率很低。

## **分析器**

Mysql先进行词法解析在进行语法解析，sql就是由多个字符串和空格组成的一条sql语句。

Kundb是边进行词法解析边进行语法解析。

## **优化器**

Mysq优化器将判断你使用了哪种索引，使用了何种连接，确定效率最高的执行方案。

Kundb是确定你该发到那个地方去执行速度最快

## **执行器**

Mysql执行器首先判断你有没有执行这条语句的权限，有的话，打开表，根据表的引擎定义，使用这个引擎的接口执行。

Kundb执行器就是executor函数，有权限后，就引用底层mysql的接口去执行。

# Sql的执行顺序

From连接：

表之间做笛卡尔积，产生虚表VT1(虚表其实就是视图，数据来自多张表的执行结果）

ON过滤：

对From连接的结果进行ON筛选，产生VT2

JOIN连接：

如果是内连接，那么From已经默认做了，如果是外链接，那么就要加入外部行，注意， mysql不支持全外连接。产生VT3

WHERE过滤：

对上一步的虚拟表进一步筛选，生成VT4

Where和on的区别：

如果有外部列，ON针对过滤的是关联表，如left join过滤的就是右侧表

无外部列时，两者毫无区别

GROUP by：

对VT4进行分组操作，产生虚拟表VT5

Having：

针对分组过滤，产生VT6  
SELECT：

按照SELECT进行筛选，生成VT7

DISTINCT：

去重，但如果运用了group by，那么去重是多余的，生成VT8

ORDER by：

排序VT8，返回一个游标，而不是虚拟表。

# 临时表

临时表就是在执行SQL语句的过程中，通常会创建一些存储中间结果集的表，临时表只对当前连接可见，在连接关闭时，临时表会被删除并释放所有表空间。

分为两种：

一是内存临时表：使用memory存储引擎

二是磁盘临时表：使用MyISAM存储引擎

会产生临时表的几种情况：

1. UNION查询
2. Order by和group by字句不一样时，因为分组后，只能看到分组的那一列和聚合函数，很明显如果那一列不是order by指定的索引列，会创建临时表再次排序。
3. 子查询

# B树

B树的主要目的是减少磁盘的IO操作，一般远小于logn。那既然这样，为啥还要红黑树和AVL树呢，原因我认为是维护不易，页分裂和页合并是二叉树不具备的劣势。

## 插一句：红黑树和AVL树的区别：

红黑树的查询略逊于AVL树，但是红黑树的维护要简单一点，也就是多了着色的操作，那么每次插入和删除的平均旋转次数远小于AVL树。

## 在插一句：B树和B+树对磁盘极度友好，而对内存可能友好的不明显：

树矮胖，避免寻道次数，降低IO

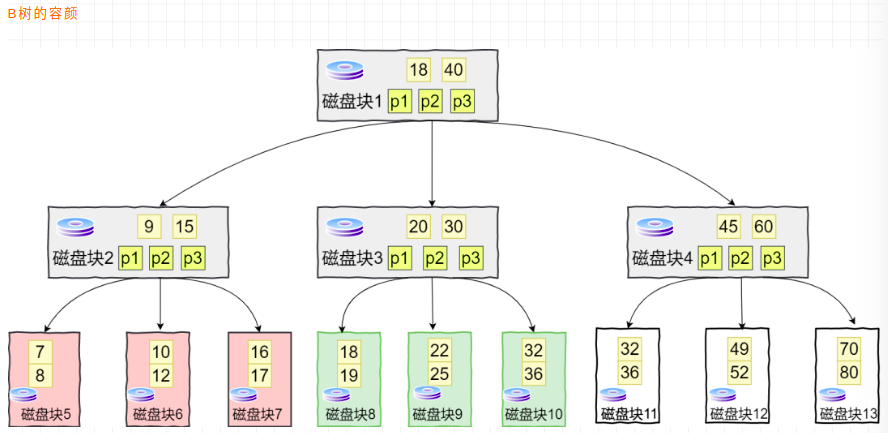
## 再插一句：B+树相比B树的优势：

1. 查询效率更稳定，都是Ologn
2. 遍历所有叶子节点即可实现整棵树便利，其他属性结果，需要中序遍历达到相同效果。
3. 更矮胖，查询更快。

## B树的特性

M阶B树

1. 除根节点和叶子节点外，每个节点有t个关键字，t+1个孩子节点（ceil(M/2) / 2 < t < M)。根节点的儿子数量范围为[2, M].
2. 每个节点都会存数据，关键字是key，也就是说val也会存在节点上。
3. 关键字升序排列
4. 叶子节点位于同一层



## **B+树的特性**

1. 孩子数量 == 关键字数量（原因是非叶子节点不存数据，所以非叶子节点只存key，不存val）
2. 所有关键字都存在叶子节点，构成双向有序链表。

## **页分裂操作**

当插入一个值后，超过了mysql页的大小，则会将中间节点上移到父节点，也称为成长，然后分裂。

