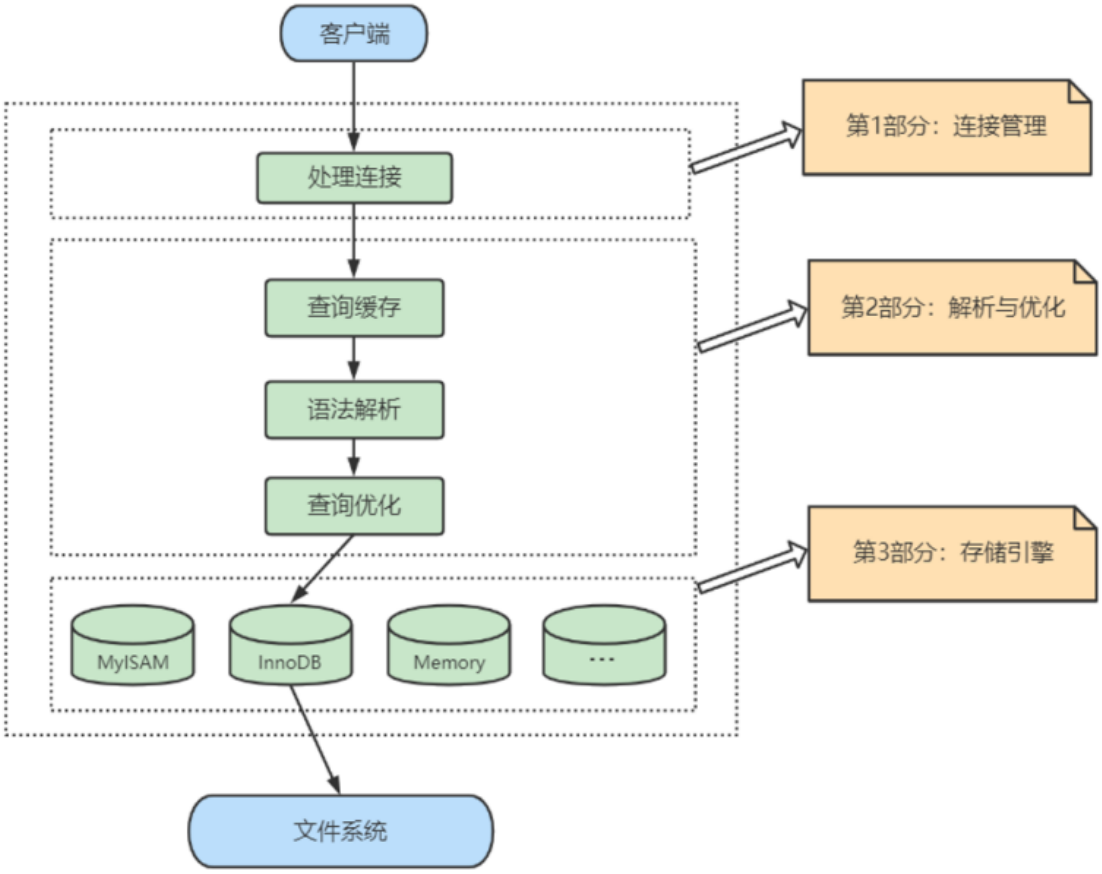
1，逻辑架构

数据库与数据库实例

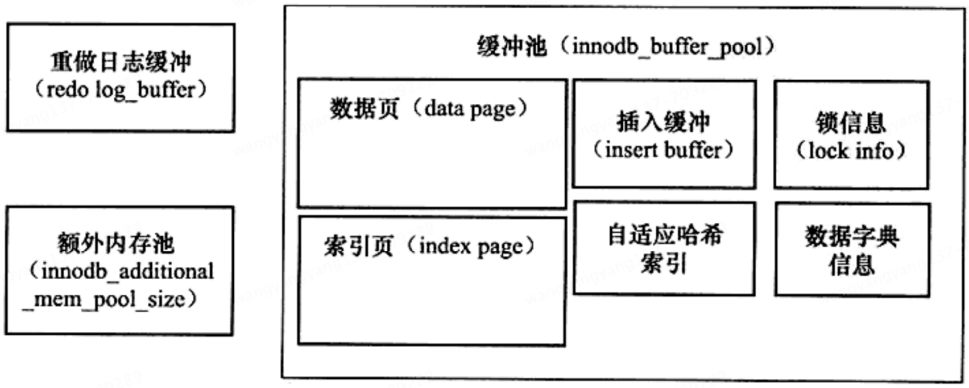
- 数据库：数据库时文件的集合，是依照某种数据模型组织起来并存放于耳机存储器中的数据集合
- 数据库实例：数据库实例是程序，是位于用户与操作系统之间的一层数据管理软件，用户对数据库数据的任何操作，报错数据库定义、数据查询、数据维护、数据库运行控制等都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库打交道

1.1 逻辑架构



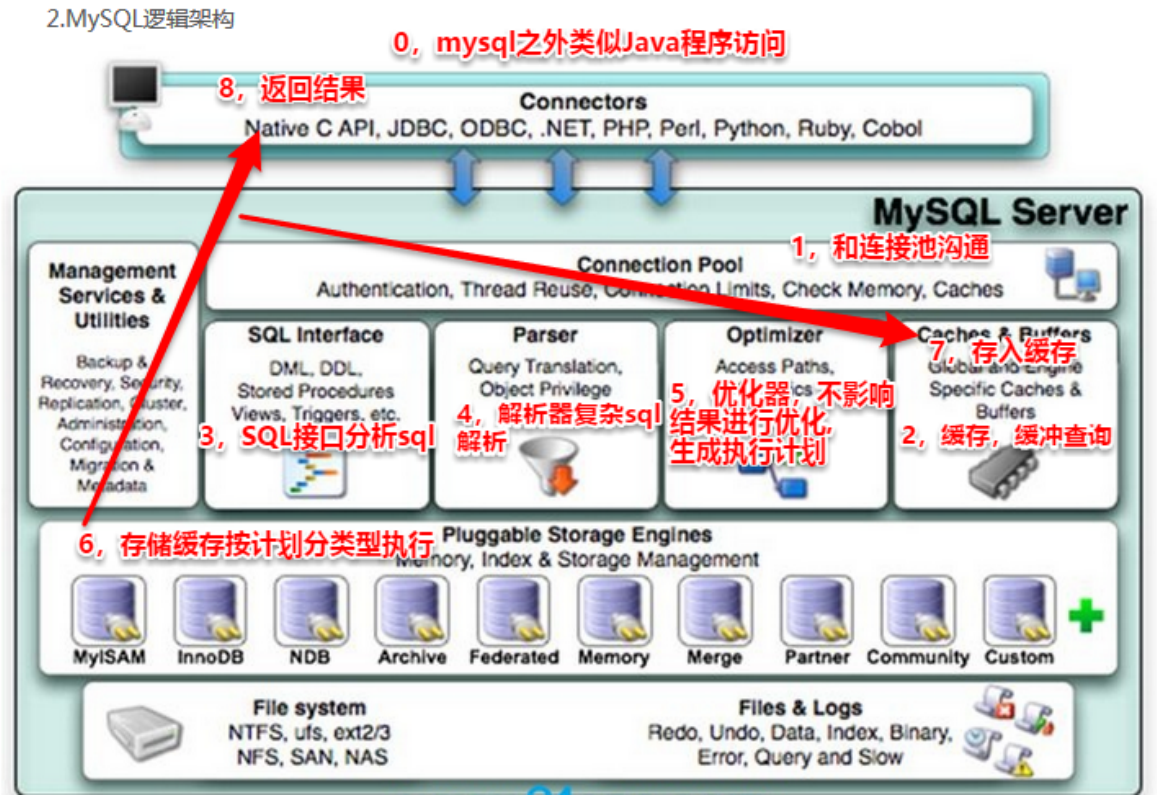
- 连接层：客户端访问服务器之前，做的第一件事就是建立TCP连接（**TCP发送数据最耗时**），并检测账号密码是否正确，之后再连接池中给该客户端分配一个连接；
- 服务层：服务层的主要作用是提供和SQL交互的接口，解析并检查SQL是否语法正确，优化分析，并且使用缓存。
    - 缓存组件：查询到来时先到缓存中进行查找，没有的话去表中查找，并将结构保存在缓存中，**缓存可在不同的客户端之间共享**。mysql8.0删除了缓存
    - 解析器：会将传入的sql解析，检查语法是否正确，同时会检查该用户是否有对应的权限
    - 优化器：优化器会根据所创建的索引，对sql进行优化
- 存储引擎层：MySQL的存储引擎是插件式的，他真正的负责了对mysql中数据的存储和提取，对物理磁盘上的数据进行操作。不同的存储引擎具有不同的功能
- 存储层：所有的数据都（表结构，用户信息，每一行内容，索引等）都是以文件系统的方式存储在物理磁盘上的。

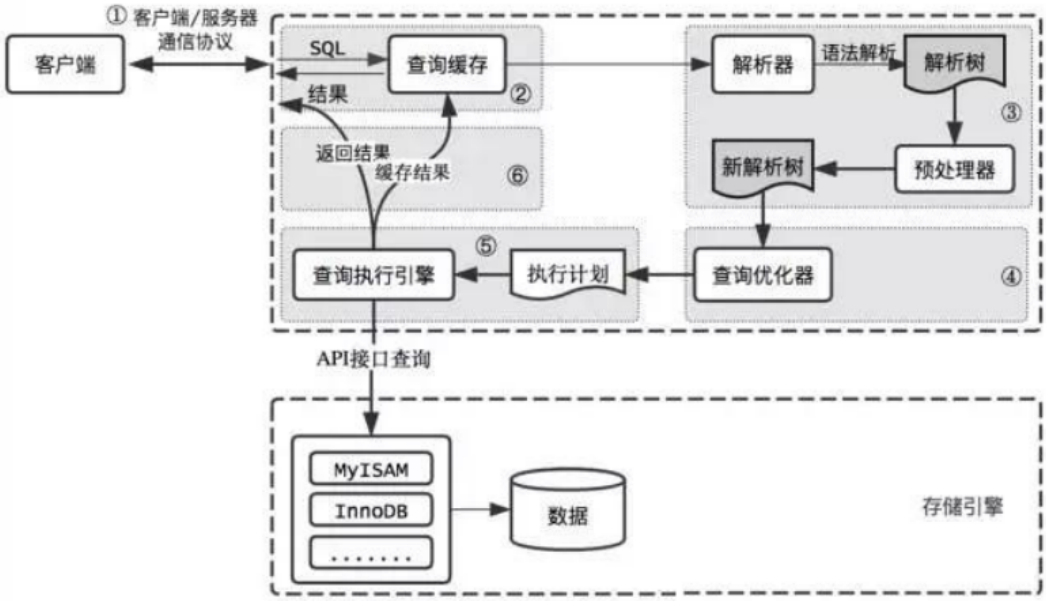
- **缓存池**
  - 缓存池不能单纯的认为是只缓存了数据页和缓存页。缓存池中有：索引页、数据页、undo页。插入缓冲、自适应哈希索引、InnoDB存储的锁信息、数据字典信息等



1.2 SQL执行流程

5.7：SQL执行流程



1. 

2. 查询缓存：Server 如果在查询缓存中发现了这条 SQL 语句，就会直接将结果返回给客户端；如果没有，就进入到解析器阶段。需要说明的是，因为查询缓存往往效率不高，所以在 MySQL8.0 之后就抛弃了这个功能。

大多数情况查询缓存就是个鸡肋，为什么呢?

查询缓存是提前把查询结果缓存起来，这样下次不需要执行就可以直接拿到结果。需要说明的是，在 MySQL 中的查询缓存，不是缓存查询计划，而是查询对应的结果。这就意味着查询匹配的 鲁棒性大大降低，只有 相同的查询操作才会命中查询缓存 。两个查询请求在任何字符上的不同（例如：空格、注释、大小写），都会导致缓存不会命中。因此 MySQL 的 查询缓存命中率不高 。

同时，如果查询请求中包含某些系统函数、用户自定义变量和函数、一些系统表，如 mysql 、information_schema、 performance_schema 数据库中的表，那这个请求就不会被缓存。以某些系统函数举例，可能同样的函数的两次调用会产生不一样的结果，比如函数 NOW ，每次调用都会产生最新的当前时间，如果在一个查询请求中调用了这个函数，那即使查询请求的文本信息都一样，那不同时间的两次查询也应该得到不同的结果，如果在第一次查询时就缓存了，那第二次查询的时候直接使用第一次查询的结果就是错误的!

此外，既然是缓存，那就有它 缓存失效的时候 。MySQL的缓存系统会监测涉及到的每张表，只要该表的结构或者数据被修改，如对该表使用了 INSERT 、 UPDATE 、 DELETE 、 TRUNCATE TABLE 、 ALTER TABLE 、 DROP TABLE 或 DROP DATABASE 语句，那使用该表的所有高速缓存查询都将变为无效并从高速缓存中删除！对于 更新压力大的数据库 来说，查询缓存的命中率会非常低。

2. **解析器**：对SQL语句进行语法分析，语义解析

   词法分析：识别字符串中各个位置的含义，即关键字

   语义解析：或者说语法分析，判断当前输入的SQL语句是否符合MYSQL语法

3. **优化器**：

   SQL查询需要使用对当前SQL语句进行优化，生成最优的方案。

4. **执行器**：

   首先判断用户是否有权限，如果没有，则权限错误。否则执行SQL。

SQL 语句在 MySQL 中的流程是：SQL语句→查询缓存→解析器→优化器→执行器。

4 数据库文件

- 参数文件：mysql数据启动时，需要读取的文件信息，存储初始化参数信息、数据库文件位置、某种内存结构的大小设置
- 日志文件：redo.log 日志、undo.log日志，慢查询 日志文件等
- mysql表结构文件：以.frm结尾的后缀存储表结构
- socket文件：使用unix域套接字方式进行连接时需要的文件
- pid文件：MySQL实例的进程ID文件
- 存储引擎文件：MySQL 表存储引擎的关系，每个存储引擎都会有自己的文件格式存储数据。这些存储引擎存储了记录和索引等数据

参数文件

MySQL实例启动时，数据库程序实例会先读取一个参数文件，寻找数据库各个文件的位置以及初始化参数信息

错误日志

- 执行错误的异常会记录在err文件中

慢查询日志

- 查询慢的语句会记录在慢查询日志中，设置条件：
  - 指定慢查询记录时间阈值， SQL执行超过此时间则会被记录到日志
  - 指定sql语句未使用索引开关，若sql语句未使用索引，则进行记录。
  - 指定慢查询且未使用索引数量
- MySQL 5.1 慢查询会记录在 slow_log 表里面，需设置log_output字段设置，可指定为File（文件）或Table表

查询日志

- 查询语句会放在查询日志中

二进制日志

- 所有的更新操作都会记录在bin log日志中，即使一个更新语句未改变数据，也会被记录。bin log 日志是二进制日志用户不可读。
- 二进制日志用途：
  - 恢复：通过二进制文件进行恢复数据
  - 复制：主从复制，也是通过bin log日志
  - 审计：用户通过bin log 日志的信息，判断是否有对数据库注入的共计。
- 默认未开启bin log 日志，**开启会有1%的性能损耗【官方手册指出，可以接受】**
- 开启事务时，未提交的bin log 日志会记录到一个缓存中，提交后写入bin log 日志文件中。可指定该缓存大小，每个线程开始事务时都会占用该大小的缓存，**所以不建议大事务。**

    - bin log 日志不会每次写的时候都刷盘。可以指定写缓存多少次之后可以刷盘。
- **PID文件：**
    - mqsql数据库实例启动时，会将自己的进程ID写入到pid文件。
- **库表结构文件 .frm**
    - 每张表都有与之对应的frm文件
-

5，搜索引擎

5.1简介

- 搜索引擎负责对表的数据进行读取和写入工作。**我们将不同的表设置不同的引擎**，一个库可以使用不同引擎，换句话说，**一个表使用一个引擎。**
- 再mysql5.5之后，InnoDB就是默认的存储引擎。

5.2 InnoDB引擎

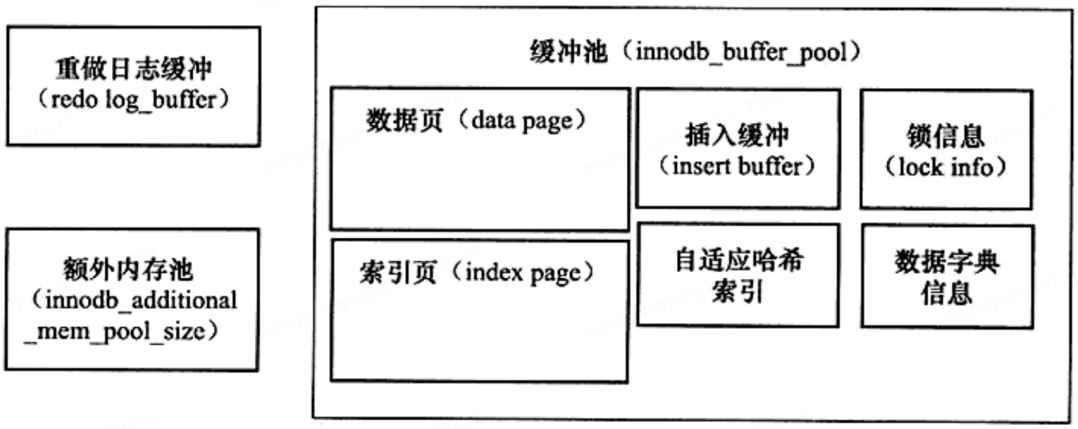
5.2.1 特点

InnoDB引擎 具备外键支持功能的事务存储引擎

- MySQL 5.5版本之后，默认采用InnoDB引擎
- **支持外键，行级锁**
- InnoDB引擎是事务存储引擎，可以处理事务，所以可以确保事务的完整提交和回滚。
- **数据库异常崩溃后可安全恢复**
- InnoDB是 为**处理巨大数据量的最大性能设计** 。换句话说，如果是处理小表的话MyISAM效果更好。
- **对比MyISAM的存储引擎， InnoDB写的处理效率差一些 ，并且会占用更多的磁盘空间以保存数据和索引。（因为索引的问题，所以在写处理方面略差）**
- **MyISAM只缓存索引，不缓存真实数据；InnoDB不仅缓存索引还要缓存真实数据， 对内存要求较高 ，而且内存大小对性能有决定性的影响。（索引方式不同）**
- InnoDB存储引擎的底层的文件结构：
    - 以.frm为后缀的文件存储表结构
    - 以.ibd为后缀的表存储数据和索引

5.2.1 缓存

缓存池：缓存池不能单纯的认为是只缓存了数据页和缓存页。缓存池中有：索引页、数据页、undo页。插入缓冲、自适应哈希索引、InnoDB存储的锁信息、数据字典信息等



- LRU列表：管理已读取的页
  - 缓冲池的内存区域，通过LRU(最近最少使用算法)进行管理。最频繁的放在LRU列表的前端，最少使用的放在LRU的尾端。在缓存池不能存放新读取的页时，将首先释放尾端的页。InnoDB引擎有进行优化，LRU加入了midpoint位置，新访问的页放在midpoint位置，而不是末端。默认midpoint位置是5/8。避免所以或数据的扫描操作，导致热点数据被刷出。
- Free列表：空闲表
  - 数据库启动时，LRU列表为空，所有页放在Free列表。分页时，查询Free列表，将该页从Free列表删除，放入LRU列表。
- Flush列表：脏页列表
  - LRU列表中的页被修改，则该页为脏页（缓存池和磁盘数据不一致）。脏页即在LRU列表，也在Flush列表。将Flush列表通过checkPoint机制刷新到磁盘。

5.2.3 插入缓冲

Insert Buffer 并不是单独的缓存信息，跟数据页一样，物理页的一部分。**使用条件：非聚簇索引更新，且索引不唯一**，后续InnoDB进行了功能拓展，变更为Change Buffer，但实际上是就将Insert，delete，update都进行了缓存。

起因：

插入数据根据主键进行递增的，聚簇索引所以一般也是递增的，数据页中的行记录也会按主键顺序存放，不需要磁盘的随机读取，速度会快一些。若主键类型是UUID这种，随机数，那么即使自增也会和非聚簇索引一样是随机的。

插入数据时，插入操作，数据还是按主键顺序插入，但非聚簇索引叶子的节点的插入不是顺序的，需要离散的访问，势必影响性能。

特点：

1. 对于非聚簇索引的插入或者更新操作，并非一次性的更新索引页中，
   - 而是先判断需要更新的索引页是否在缓存池中，若在，则直接更新，不在，则先放在Insert Buffer对象中。

2. 然后再以一定的频率和情况将Insert Buffer和非聚簇索引页进行merge。即一个更新索引页时，将多个插入节点更新合并，提高性能。

为什么要索引不唯一?

- 若索引唯一，在插入索引时，需要判断插入数据的唯一性，这是需要全表扫描，或者这个非聚簇索引的扫描，与插入缓冲的起因相同，使得该操作失去意义。

若是写密集的情况，那么Insert Buffer最大可以占到1/2的缓冲池。

Change Buffer包括Insert Buffer，update Buffer，Delete Buffer。其他与Insert Buffer操作一致。

1. 更新操作，是先对该条记录进行标记
2. Merge Buffer到索引时，将该条记录真正删除

Merge Insert Buffer:

插入缓冲合并到非聚簇索引页的情况

- 非聚簇索引页被读取到缓存池中时
- Insert Buffer BitMap追踪该非聚簇索引无可用空间时
- 主线程每秒霍进行一个刷盘，将索引信息合并到非聚簇索引页中

5.2.4 两次写操作

若数据写入磁盘时，部分已经写入，发生宕机，此时可以通过redo log日志进行恢复。doubleWriter：可以先通过数据页的副本还原该页，然后再通过redo log日志进行重做。

5.2.5 自适应哈希索引

自适应哈希索引：哈希查找速度会特别快。查询B+树相较于哈希还是慢一些的。InnoDB会监控表上索引页的查询，如果观察到建立建立哈希索引能够性能提升，则建立哈希索引。读取和写入速度可以提高2倍，默认开启自适应哈希索引。

自适应哈希索引并不是对整张表构建哈希索引，会对某些热点数据建立哈希索引。

要求:

- 对该页的查询条件一致，才可以建立哈希索引。若 where a = "xxx" 和where a = "xxx" and b = x，两个查询条件交替执行，则InnoDB引擎不会对该页建立哈希索引。
- 同一查询条件查询了100次
- 同一查询条件 查询 了N次，N = 该页记录*1/16

5.2.6 刷新临近页

当刷新一个脏页时，InnoDB迎请检测该页所在区的所有页，将临近的所有脏页也刷盘。好处是：将原本多个IO操作改为一个IO操作。

5.3 MyISAM引擎

- 5.5之前默认的存储引擎
- MyISAM 不支持事务、行级锁、外键 ， 有一个毫无疑问的缺陷就是 **崩溃后无法安全恢复**
- 只缓存索引，不缓存真实数据
- 优势是访问的 **速度快 ， 对事务完整性没有要求或者以SELECT、INSERT为主的应用**
- 应用场景：**适用于读和插入操作较多的情况选择MyISAM**
- 底层存储的文件结构：
    - 以.frm结尾的后缀存储表结构
    - 以.MYD结尾的后缀存储数据
    - 以.MYI结尾的后缀存储索引

InnoDB引擎与MyISAM引擎的区别：

首先对于InnoDB存储引擎，提供了良好的事务管理，崩溃修复能力和并发控制。因为InnoDB存储引擎支持事务，所以对于要求事务完整性的场合选择InnoDB，比如数据操作除了插入和查询，还有更新和

对比项	MyISAM	InnoDB
外键	不支持	支持
事务	不支持	支持
行表锁	表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作	行锁，操作时只锁某一行，不对其它行有影响，适合高并发的操作
缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响
自带系统表使用	Y	N
关注点	性能：节省资源、消耗少、简单业务	事务：并发写、事务、更大资源
默认安装	Y	Y
默认使用	N	Y

- **InnoDB数据库异常崩溃后可安全恢复**
- InnoDB是 为**处理巨大数据量的最大性能设计** 。换句话说，如果是处理小表的话MyISAM效果更好。
- **对比MyISAM的存储引擎， InnoDB写的处理效率差一些 ，并且会占用更多的磁盘空间以保存数据和索引。（因为索引的问题，所以在写处理方面略差）**
- **MyISAM只缓存索引，不缓存真实数据；InnoDB不仅缓存索引还要缓存真实数据， 对内存要求较高 ， 而且内存大小对性能有决定性的影响。（索引方式不同）**
- MyISAM索引方式是非聚簇索引，InnoDB包含一个聚簇索引
- MyISAM引擎每次查找都需要回表

- InnoDB数数据文件本身就是索引文件，而MYISAM索引文件与数据文件是分离的，索引文件保存数据的地址
- InnoDB中，非聚簇索引的叶子节点的data域保存的是数据主键，而MYISAM保存的是数据地址。可以直接读取数据，InnoDB需要两次查找操作，在时间上，MyISAM更快。

6，索引的数据结构

1，索引

索引（Index）是帮助MySQL高效获取数据的数据结构

索引的本质：排好序的快速查找数据结构

优点：

- 降低数据库的IO操作，节省时间
- 创建唯一性索引，来保证数据库的表中每一行数据的唯一性
- 在实现数据的参考完整性方面，可以 加速表和表之间的连接 。
- 减少查询中分组和排序的时间 ，降低了CPU的消耗。

缺点：

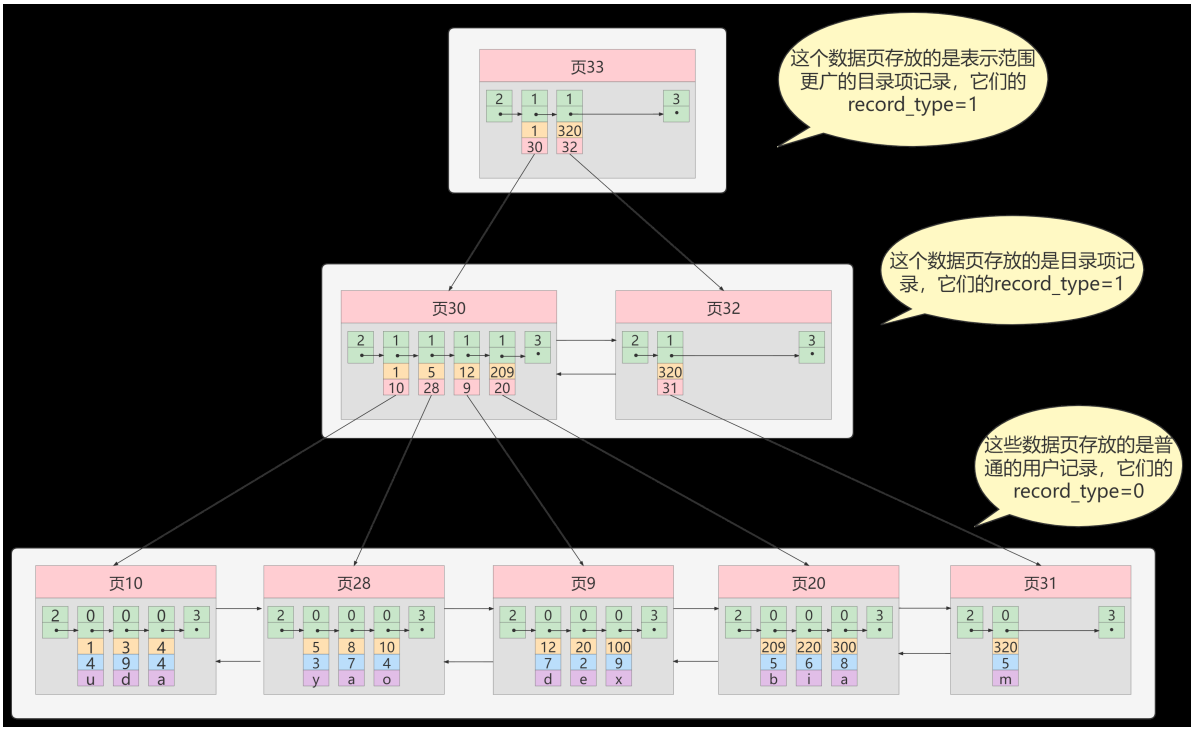
- 建立和维护索引需要时间
- 缓存索引需要空间存储
- 索引值提高了查询速度，同时降低了更新表的速度，对表数据的修改也需要更改索引，其他操作时间延长。

2，InnoDB中索引

B+树索引无法找到一个给定主键值的具体行数据。B+树只能找到被查找数据锁在的页，然后将该页读取到内存中，在从该页中查找，得到最后的数据。

一般情况下，我们用到的B+树都不会超过4层，因为层数越少，IO次数越少，这样查询的时间就越少。

InnoDB中，索引按不同的物理方式可以分为**聚簇索引**和**非聚簇索引**（二级索引，辅助索引）



- **聚簇索引**
  - 特点　（页即非叶子节点）
    - 页内的记录存储页号和主键， 叶子节点 存储的是完整的用户记录
    - 同一层级的页使用双向链表，不同层级的页使用单向链表
  - 优点：
    - 数据访问更快，从聚簇索引中获取数据比非聚簇索引更快。
    - 适用于**排序查找**和**范围查找**
  - 缺点：
    - 插入速度严重依赖于插入顺序，更新主键的代价很高

MySQL中，只有InnoDB引擎支持聚簇索引，而且默认创建，MyISAM并不支持

每个表只能有一个聚簇索引，MySQL会使用主键自动构建索引

如果表没有定义主键，InnoDB会选择非空的唯一索引代替，如果没有，MySQL则会隐性的构建一个

- **非聚簇索引**

（若搜索条件是主键查找，则使用聚簇索引，**若是其他条件查找，则需要使用非聚簇索引，本质是构建多颗B+树，以表另外一字段列作为索引，不使用id。但是叶子节点数据是id。在查找到id后，再从聚簇索引中查询完整的用户信息**）

概念：回表 我们根据这个以c2列大小排序的B+树只能确定我们要查找记录的主键值，所以如果我们想根据c2列的值查找到完整的用户记录的话，仍然需要到 聚簇索引 中再查一遍，这个过程称为 回表 。也就是根据c2列的值查询一条完整的用户记录需要使用到 2 棵B+树!

为什么我们还需要一次 回表 操作呢？直接把完整的用户记录放到叶子节点不OK吗？

答：叶子节点也保存数据会造成空间的浪费（因为保存了多份相同的数据，而且索引未必是唯一的）。而且以其他条件作为索引，可能不是非空，唯一。

- 联合索引

  （本质也是非聚簇索引，选择了多个列作为索引）

    - 联合索引指的是：选择C1，C2多个列作为索引，构建一颗B+树。
    - 先把各个记录和页按照c1列进行排序，若C1相同，则再按C2排列。叶子阶段也是包含主键数据

B+树的插入

B+树的插入必须保证插入后叶子节点的记录依然有序。

- 非叶子节点、叶子节点没满
    - 细节插入到叶子节点
- 非叶子节点没满、叶子节点满
    1. 拆分叶子节点
    2. 中间的节点放在非叶子节点
    3. 小于中间节点的放左边，大于等于的放右边
- 非叶子节点满、叶子节点满
    1. 拆分叶子节点
    2. 中间节点放在非叶子节点，小于中间节点的放左边，大于等于的放右边
    3. 拆分非叶子节点
    4. 小于中间节点的记录放左边，大于等与的放右边
    5. 中间节点放在上一层的非叶子节点

数据的插入并非只能拆分页的操作，也会像平衡二叉树的旋转操作，将原页数据分到其他页中，而没有生成新页面。来减少一次页的拆分操作，提高性能。

B+树的删除

B+树的删除使用填充因子控制树的变化。默认值是50%。叶子节点的容量节点数占用页节点空间小于50%时，缩容。

- 叶子节点大于填充因子，中间节点大于填充因子：将节点直接从叶子节点删除，若该节点页数非叶子节点的信息，将该节点的右节点代替
- 叶子节点小于填充因子，中间节点大于填充因子：合并叶子节点和它的兄弟节点，同时更新中间节点
- 叶子节点小于填充因子，中间节点小于填充因子：
    1. 合并叶子节点和它的兄弟节点
    2. 更新非叶子节点
    3. 合并非叶子节点和它的兄弟节点

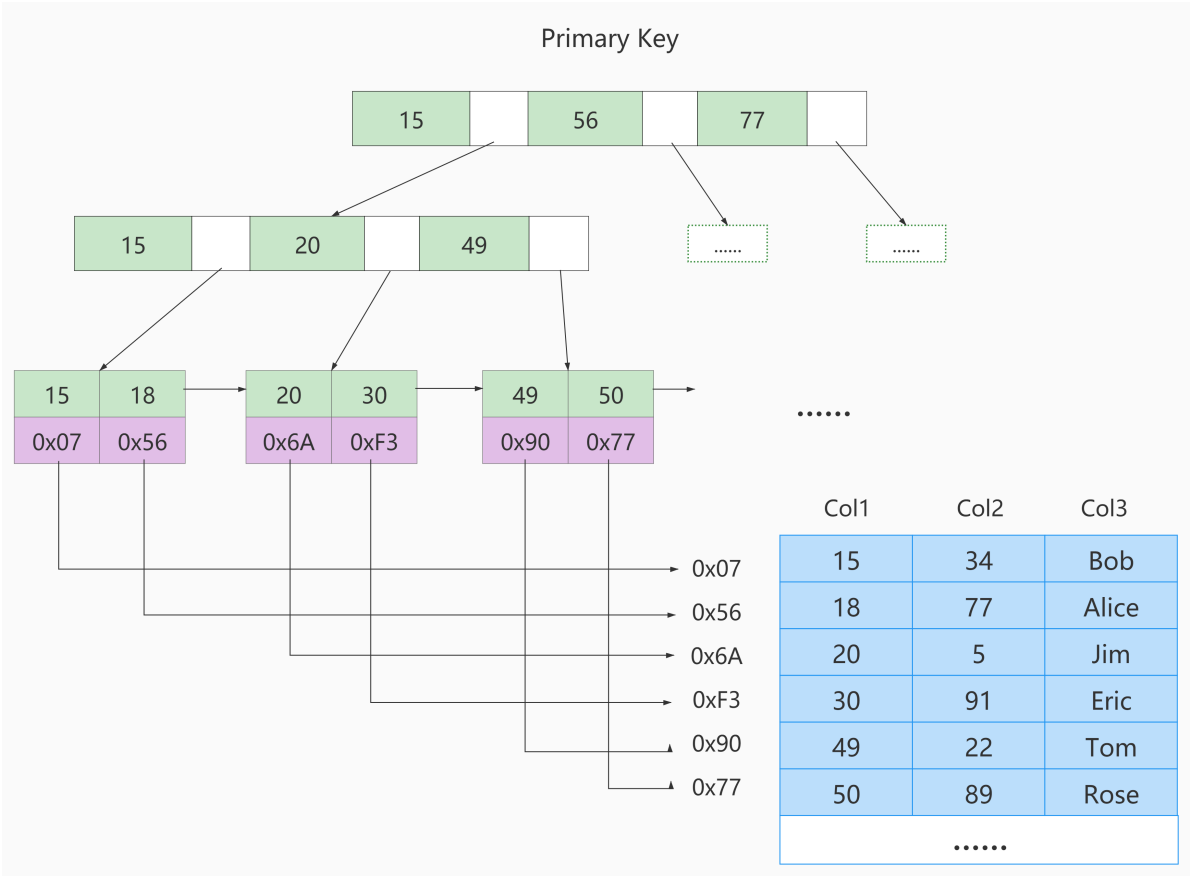
InnoDB的B+树索引的注意事项

1. 根页面位置万年不动
2. 内节点（页目录）中目录项记录的唯一性，
   非聚簇索引构建B+树，也会保存主键的值，为的是避免出现页目录中，有两项完全相等的情况
3. **一个页面最少存储2条记录**

3，MyISAM中的索引

MyISAM引擎使用B+树作为索引结构，叶子节点的data域存放的是**数据记录的地址**

- MyISAM原理
  - 使用的都是非聚簇索引，因为索引和数据是分离的，叶子节点存储的是数据地址。



MyIsam索引和InnoDB索引的对比

- MyISAM索引方式是非聚簇索引，InnoDB包含一个聚簇索引
- MyISAM引擎每次查找都需要回表
- InnoDB数数据文件本身就是索引文件，而MYISAM索引文件与数据文件是分离的，索引文件保存数据的地址
- InnoDB中，非聚簇索引的叶子节点的data域保存的是数据主键，而MYISAM保存的是数据地址。可以直接读取数据，InnoDB需要两次查找操作，在时间上，MyISAM更快。

4， 索引的代价

索引是个好东西，可不能乱建，它在空间和时间上都会有消耗：

- **空间上的代价**

  每建立一个索引都要为它建立一棵B+树，每一棵B+树的每一个节点都是一个数据页，一个页默认会占用 16KB 的存储空间，一棵很大的B+树由许多数据页组成，那就是很大的一片存储空间。

- **时间上的代价**

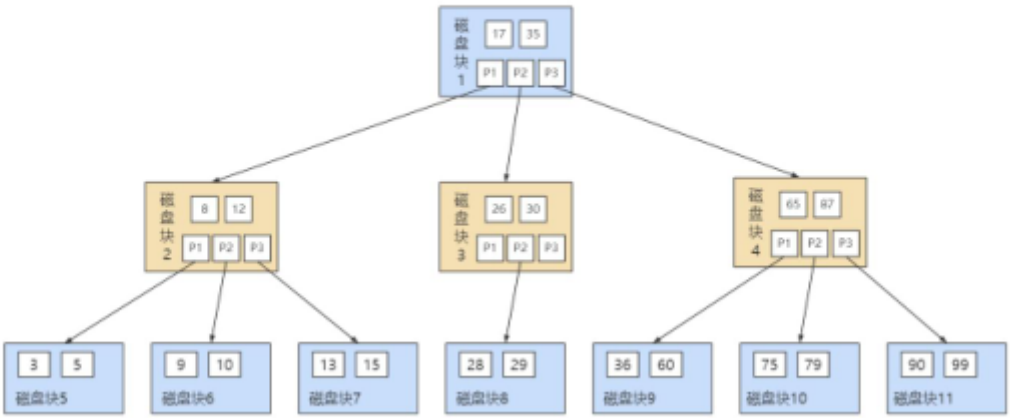
每次对表中的数据进行 增、删、改 操作时，都需要去修改各个B+树索引。而且我们讲过，B+树每层节点都是按照索引列的值 从小到大的顺序排序 而组成了 双向链表 。不论是叶子节点中的记录，还是内节点中的记录（也就是不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单向链表。而增、删、改操作可能会对节点和记录的排序造成破坏，所以存储引擎需要额外的时间进行一些 记录移位 ， 页面分裂 、 页面回收 等操作来维护好节点和记录的排序。如果我们建了许多索引，每个索引对应的B+树都要进行相关的维护操作，会给性能拖后腿。

5，B树与B+树，Hash索引，AVL索引

这一章节为MySQL数据结构选择的合理性，目的是减少磁盘的IO次数

6.5 B-Tree

B 树的结构如下图所示：



- 特点：
  - 叶子节点和非叶子节点都存放数据，搜索可能到不了叶子节点就结束
  - B树在插入和删除结点的时候导致树不平衡，则通过自动调整叶子节点的位置保持平衡
  - 搜索性能等价于二分查找

B+ 树和 B 树的差异：

- B+ 树中，非叶子节点仅用于索引，不保存数据记录，跟记录有关的信息都放在叶子节点中。而 B 树中， 非叶子节点既保存索引，也保存数据记录 。
- B+树的查找效率更高，而且更加稳定，因为都需要多次索引才会读取到数据，而B树可能在非叶子节点就读取到数据，虽然说这样看似效率更高，但是非叶子节点存放 数据，势必存放的索引减少，非叶子节点增多，增加IO，查找效率就降下来了。
- **B+树同级之间存储双向链表。**

Hash索引结构　只有Memory存储引擎中使用

hash查询速度很快，但是有缺点

- **hash索引在等值判断的情况下是速度很快的。**如果进行范围查询，则退化为全表查询
- 范围查询，分组查询失效，退化为全表查询
- 无法进行联合查询。因为不同的字段结合在一起，无法判断。【】
- 若数据值重复太多，非常多的哈希碰撞。查询时 非常耗时

二叉搜索树

开始选择二叉搜索树，可是二叉搜索树容易出现斜树。IO数量很多，所以选择平衡二叉搜索树

AVL树 平衡二叉搜索树

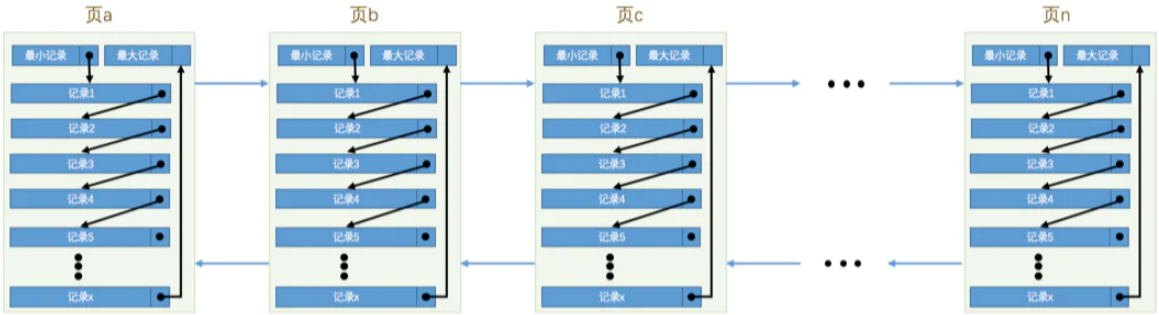
- 依然很多IO操作，可以将二叉树改为M叉树--->B 树

7，InnoDB数据存储结构

InnoDB存储结构包含：段、区和页。

- 段：包含数据段、索引段、回滚段
  - 数据段是主键B+树的叶子节点信息
- 区：连续页组成的空间，一个区默认有64个连续页（可设置数量）
- 页：基本单位，16kb

1. 数据库的存储结构：页



- InnoDB 将数据划分为若干个页，InnoDB中页的大小默认为 16KB 。
- **数据库管理存储空间的基本单位是页**
- 各个页之间可以不在物理结构上相连，靠双向链表连接。
- 页内部的数据通过单向链表连接，每个数据页都会存储一个页目录，可以通过二分法进行查找

InnoDB行记录格式

看 杂七杂八.md 中的InnoDB行格式。InnoDB分为Compact和Redundant格式。

8，索引的创建与设计原则

1，索引的声明与使用

1，分类：

- 逻辑功能角度：普通索引，唯一索引，主键索引和全文索引
- 物理实现角度：聚簇索引和非聚簇索引
- 字段个数角度：单列索引和联合索引

InnoDb引擎支持B+树索引、全文索引和哈希索引（自适应，无需人为干预，自动为表生成哈希索引）。

B+树索引无法找到一个给定主键值的具体行数据。B+树只能找到被查找数据锁在的页，然后将该页读取到内存中，在从该页中查找，得到最后的数据。

2，创建

创建表时，在声明有主键约束，唯一性约束，外键约束的字段上，会自动创建相关索引

- 创建表时：

```
CREATE TABLE table_name [col_name data_type] [UNIQUE | FULLTEXT | SPATIAL]
[INDEX | KEY] [index_name] (col_name [length]) [ASC | DESC]
```

- 表已经存在，添加索引

```
ALTER TABLE table_name ADD INDEX index_name(`索引字段1`,`索引字段2`);
```

- 删除索引

```
ALTER TABLE table_name DROP INDEX index_name;
```

- 添加字段

```
alter table `table_name` add column '字段名' varchar(30)字段类型  not null #约束（唯一性，不为null，默认值）
```

- 删除字段

```
alter table `table_name` drop column '字段名'
```

2，索引的设计原则

1. 字段的数值有唯一性的限制，可以设置为索引
2. GROUP BY 和 ORDER BY 的列
3. 经常作为，增，查，改操作的WHERE 查询条件
4. 经常去重字段需要创建索引

5. 在条件允许的情况下，可以使用联合索引，优先将筛选粒度大的放在左侧。**联合索引优于单值索引**
6. 多表join连接操作时，创建索引注意事项：
    1. 连接表的数量不要超过3张
    2. 对where条件创建索引
    3. 对用于连接的字段要创建索引
7. 使用列的类型小的创建索引（即对UUID建立索引，没必要）（比如说：varchar（50）varchar（20），选择小的那个，在大的在页中存储的少，则增加IO）
8. **使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间**

限制索引数目

不要过度索引。

3，不适合创建索引的情况

- where使用不到的字段，不使用索引
- 数据量小的表最好不要使用索引
- 有大量重复数据的列上不要建立索引（**区分度不够的列不要创建索引**）
- 经常更新或者修改的字段不创建索引

MySQL性能分析

- 观察服务器状态
- 开启缓存，查看缓存
- 开启慢查询
    - SQL等待时间过长 通过指令 **show profiles**
        - 调优服务器参数（加内存，刷盘策略）
    - SQL执行时间过长　通过指令 Explain+SQL
        - 进行索引优化
        - 表结构的优化
- 开启主从复制

如何发现慢SQL

开启慢查询日志（默认不开启）,修改配置。重启MySQL

修改conf文件

配置说明：

```
slow_query_log = ON
long_query_time = 5
slow_query_log_file = /opt/soft/mysql/log/slow.log
log_queries_not_using_indexes=on
```

slow_query_log： 开启或关闭慢查询日志。

slow_query_log_file： 指定生成的慢查询日志路径（未设置则和默认和数据文件放一起）。

long_query_time： 慢查询记录时间阈值，SQL执行超过此时间则会被记录到日志（单位：秒，默认10秒）。

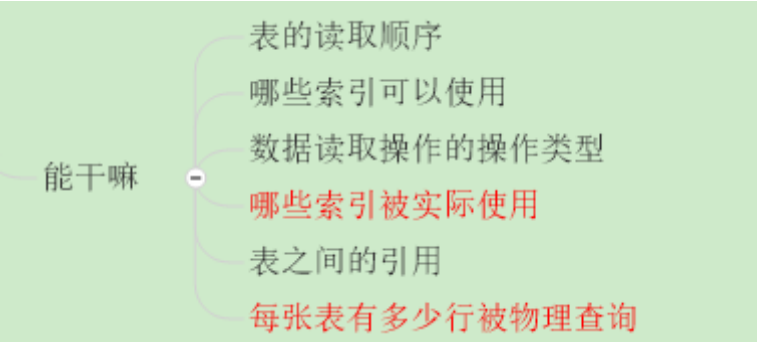
log_queries_not_using_indexes： 是否记录未使用索引的SQL。

log_throttle_queries_not_using_indexes: 每分钟允许记录到slow log 且未使用索引的sql语句次数。

4，性能分析EXPLAIN

Explain：使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈

用处：



Explain + SQL语.

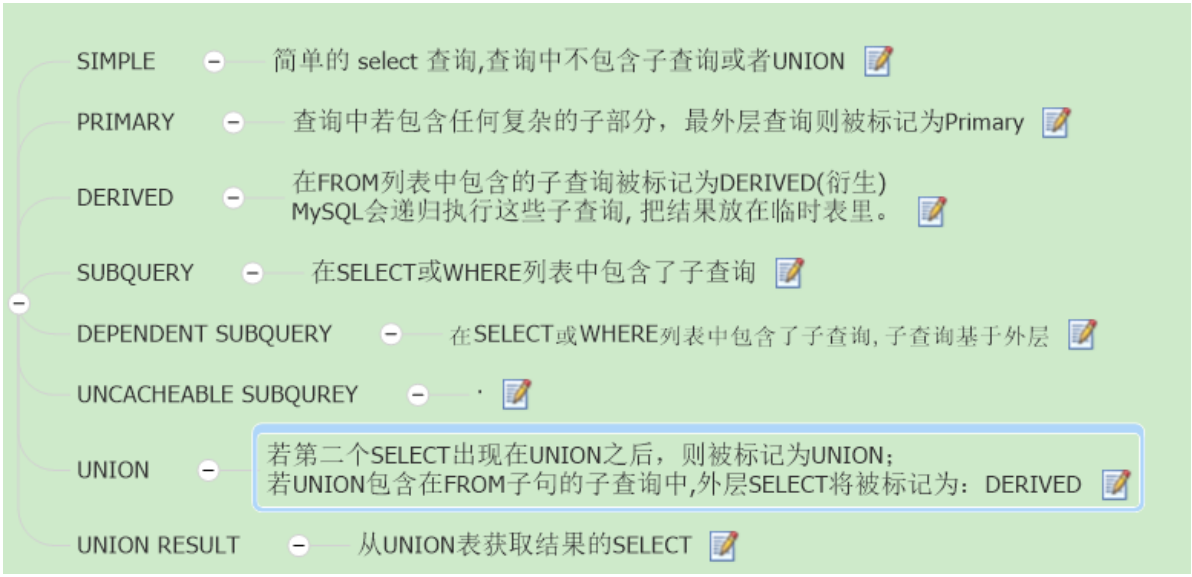
属性：

- **id： select查询顺序，表示查询中执行select子句或操作表的顺序**
  - id不同：若是子查询，id序号会递增。id越大，优先级越高。越先被执行
  - id相同：执行顺序，由上到下。
  - id同与不同，同时存在

  关注点：

id号每个号码，表示一趟独立的查询。一个sql 的查询趟数越少越好。即只有id=1为最好

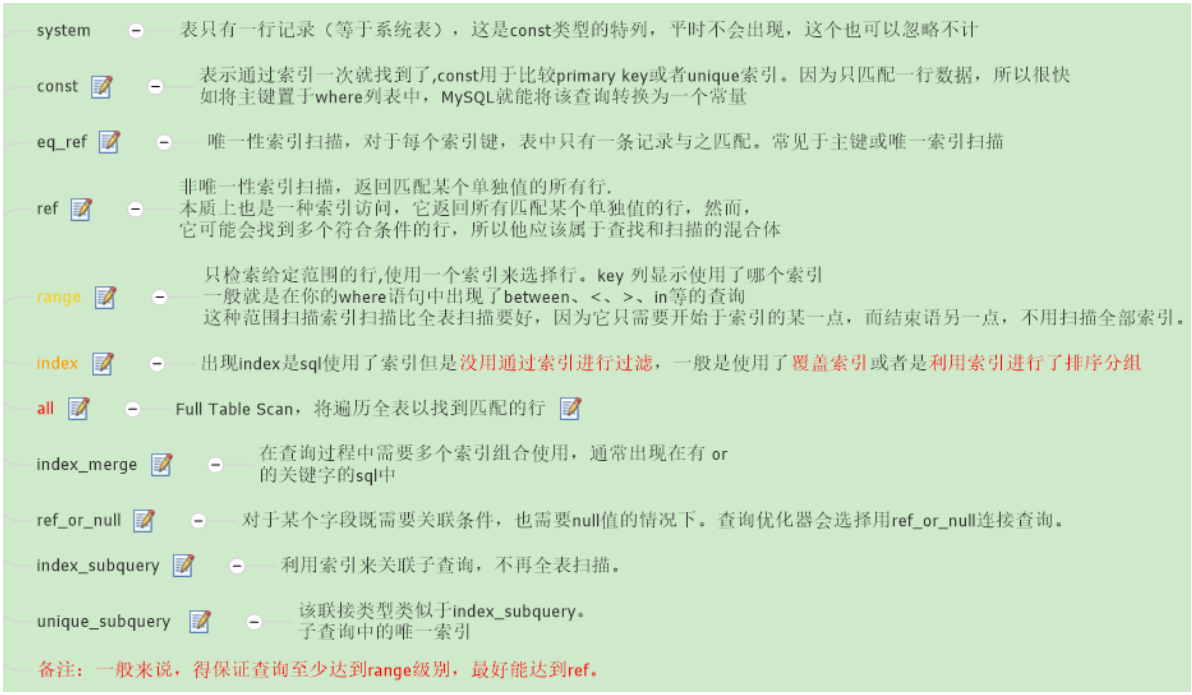
- **select_type：**查询的类型，主要是用于区别普通查询、联合查询、子查询等的复杂查询



- **type：表示查询使用了哪种类型。即查询困难程度（优化指标）**

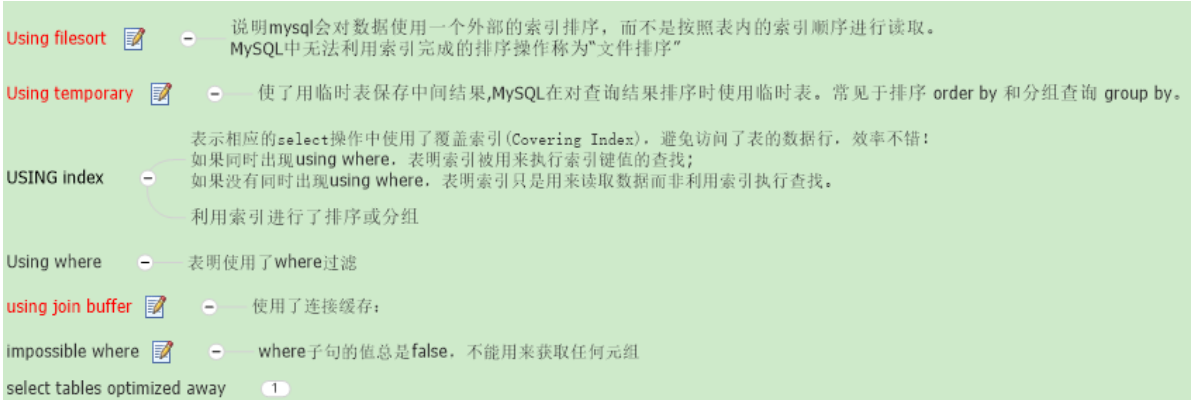
  从最好到最差依次是：

  system>const>eq_ref>ref>range>index>ALL



- possible_keys：查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用
- **keys：** 实际使用的索引。 查询中若使用了覆盖索引，则该索引和查询的select字段重叠
- **key_len:** where后面的筛选条件命中索引的长度，与order by 和group by无关。应用索引的长度。数值为索引中使用的字节数，**越大越好**

  key_len字段能够帮你检查是否充分的利用上了索引
- **rows：**rows列显示MySQL认为它执行查询时必须检查的行数。 **（越小越好）**
- Extra：包含不适合在其他列中显示但十分重要的额外信息



Using filesort：没有应用到索引进行排序，很慢。

Using temporary： 应用到order by 或者 group by。很慢很慢，一般出现Using temporary，会连带着出现

Using filesortUsing join buffer：join 连接条件key未使用索引，很慢

impossible where：where 条件逻辑错误。

5，索引失效的情况

- 不满足**最佳左前缀原则**，没有按照联合索引的顺序查询，或者说缺少左侧列的条件查询
  索引文件具有 B+-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。
- **索引列使用函数或者计算**（计算、函数、(自动or手动)**类型转换**）会导致索引失效
- 联合查询，范围查询后的索引列无效。（**构建联合索引时，将经常范围查询的列（字段）放在最后**）
- 字符类型转换也会索引失效
- **mysql使用不等于**（`!=`或者`<>`），索引失效
- is null可以使用索引，is not null 不能使用 。 **不等于当然是失效的**
- **like以通配符开头('%abc...') mysql索引失效**

6，关联查询的优化总结（join）

关联两个表的关联分为**驱动表**和**被驱动表**。驱动表作为主表，被驱动表是从表。

关联查询就是嵌套查询，会首先遍历驱动表，中选择符合的数据，在从被驱动表中进行匹配。所以是小表驱动大表。一般大表会建立索引，这样能够使得速度更快

- **永远用小结果集驱动大结果集（本质减少外层循环的数据量）**
- 保证被驱动表的JOIN字段，已经建立索引
- 内连接，MySQL会自动选择小结果集作为驱动表
- 子查询尽量不要放在被驱动表，有可能使用不到索引。
- 能够直接多表关联的尽量直接关联，不用子查询。

8.0使用Hash join

STRAIGHT_JOIN：1，指定驱动表和被驱动表 2，明确前后两表的数量级

STRAIGHT_JOIN=> inner join

STRAIGHT_JOIN前面为驱动表，后面为被驱动表。

适用于 MySQL优化器选择的驱动表不好情况下，要替换驱动表

SHOW PROFILE

查看查询成本。

```
show profiles;
```

show profile的常用查询参数：
① ALL：显示所有的开销信息。

② BLOCK IO：显示块IO开销。

③ CONTEXT SWITCHES：上下文切换开销。

④ CPU：显示CPU开销信息。

⑤ IPC：显示发送和接收开销信息。

⑥ MEMORY：显示内存开销信息。

⑦ PAGE FAULTS：显示页面错误开销信息。

⑧ SOURCE：显示和Source_function，Source_file，Source_line相关的开销信息。

⑨ SWAPS：显示交换次数开销信息

7，子查询优化

- 可以使用连接（JOIN）查询来替代子查询
- **不要使用NOT IN 或者 NOT EXISTS，用LEFT JOIN xxx ON xx WHERE xx IS NULL替代**

8，排序优化

问题：在 WHERE 条件字段上加索引，但是为什么在 ORDER BY 字段上还要加索引呢?

SQL中，在where 和order使用索引，where是避免全表扫描，order by使用索引是为了避免FileSort排序。

FileSort排序：一般在内存中排序，占用CPU过多，如果结果较大，会产生临时文件，增加IO，**效率较低**

所以：索引尽量使用 索引完成 ORDER BY 排序。

着重看Extra字段，

- Using index: 所有的请求列都在一个索引树中，无需访问实际的行记录。即索引覆盖

- Using index condition：确实命中了索引，但不是所有的列数据都在索引树上，还需要访问实际的行记录。
  - 性能不如Using index好
- Using where说明，在索引的基础上，使用了where条件过滤。
- Using filesort： 最差劲，坚决不能出现

order by

- **无过滤，不索引。**
  - 即没有limit条件，无法应用索引。
- **顺序错，必排序。**
  - 即建立索引的顺序与order 条件的顺序不同，出现 Using filesort
- **方向反，必排序**
  - 即order排序多个属性，若asc和desc都存在，则出现 Using filesort

filesort算法：

双路排序和单路排序

- 双路排序（进行两次磁盘扫描，**慢**） MySQL 4.1之前使用
  - 首先读取order by的字段进行排序
  - 在根据排序后的结果集，读取磁盘完整的数据。
- 单路排序（一次磁盘扫描）
  - 在磁盘中读取需要的列数据，选择其中字段order by排序

由于单路是后出的，总体而言好过双路

group by的原则与order by相同，唯一不同是没有过滤条件，也可以直接使用索引

9，覆盖索引

- 一个索引包含了满足查询结果的数据就叫做覆盖索引。
- **非聚簇索引的叶子节点，会保存数据的索引字段和主键。若select查询列仅为这些数据，则不必回表。**（即建索引的字段正好是覆盖查询条件中所涉及的字段）。
- **再次验证了select，where 操作是不同时运行的，只有在这种特殊情况下，才能跳出select顺序执行的逻辑。**

优缺点：

- 避免了InnoDB的二次查询，（回表）
- 可以把随机IO变成顺序IO加快查询效率
- 索引字段的维护 总是有代价的。(缺点)

10，索引下推（ICP）

MySQL5.6中的新特性，是一种在存储引擎层使用索引过滤数据的一种优化方式。ICP可以减少存储引擎访问基表的次数以及MySQL服务器访问存储引擎的次数。

实例：

```
# index name_age (age,name);创建联合索引
```

```
select * from user where name like'%A' and age = 10; #（100条数据回表）
```

由于name是模糊查询,按理来说是失效的。只会使用age=10这个索引条件，再回表进行模糊查询。索引下推（ICP）会实现。在二级索引中的name字段进行模糊查询，将结果再进行回表。提高了查找速度。

ICP的使用条件：

- 只能用于非聚簇索引，减少基表（聚簇索引构建的表）的查询次数
- 只有非聚簇索引字段为筛选字段才能起作用。即二级索引中有该字段才可以使用ICP

13，事务

ACID

何为 ACID 特性呢?

1. **原子性**（`Atomicity`）： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. **一致性**（`Consistency`）： 执行事务前后，数据保持一致，例如转账业务中，无论事务是否成功，转账者和收款人的总额应该是不变的；
3. **隔离性**（`Isolation`）： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
4. **持久性**（`Durability`）： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

数据事务的实现原理呢?

我们这里以 MySQL 的 InnoDB 引擎为例来简单说一下。

MySQL InnoDB 引擎使用 redo log(**重做日志**) 保证事务的**持久性**，使用 undo log(**回滚日志**) 来保证事务的**原子性**。

MySQL InnoDB 引擎通过 **锁机制**、MVCC 等手段来保证事务的隔离性（默认支持的隔离级别是`REPEATABLE-READ`）。

保证了事务的持久性、原子性、隔离性之后，一致性才能得到保障。

- **一致性（consistency）：**

(国内很多网站上对一致性的阐述有误，具体你可以参考 Wikipedia 对 Consistency 的阐述)

根据定义，一致性是指事务执行前后，数据从一个 合法性状态 变换到另外一个 合法性状态 。这种状态是 语义上 的而不是语法上的，跟具体的业务有关。

那什么是合法的数据状态呢？满足 预定的约束 的状态就叫做合法的状态。通俗一点，这状态是由你自己来定义的（比如满足现实世界中的约束）。满足这个状态，数据就是一致的，不满足这个状态，数据就是不一致的！如果事务中的某个操作失败了，系统就会自动撤销当前正在执行的事务，返回到事务操作之前的状态。

数据并发问题

3.2 数据并发问题

针对事务的隔离性和并发性，我们怎么做取舍呢？先看一下访问相同数据的事务在 不保证串行执行 （也就是执行完一个再执行另一个）的情况下可能会出现哪些问题：

1. 脏写（Dirty Write）

对于两个事务 Session A、Session B，如果事务Session A 修改了 另一个 未提交 事务Session B 修改过 的数据，那就意味着发生了 脏写

---

2. 脏读（Dirty Read）

对于两个事务 Session A、Session B，Session A 读取 了已经被 Session B 更新 但还 没有被提交 的字段。之后若 Session B 回滚，Session A 读取 的内容就是 临时且无效 的。

Session A和Session B各开启了一个事务，Session B中的事务先将studentno列为1的记录的name列更新为'张三'，然后Session A中的事务再去查询这条studentno为1的记录，如果读到列name的值为'张三'，而Session B中的事务稍后进行了回滚，那么Session A中的事务相当于读到了一个不存在的数据，这种现象就称之为 脏读 。

3. 不可重复读（Non-Repeatable Read）

对于两个事务Session A、Session B，Session A 读取 了一个字段，然后 Session B 更新 了该字段。 之后 Session A 再次读取 同一个字段， 值就不同 了。那就意味着发生了不可重复读。

我们在Session B中提交了几个 隐式事务 （注意是隐式事务，意味着语句结束事务就提交了），这些事务都修改了studentno列为1的记录的列name的值，每次事务提交之后，如果Session A中的事务都可以查看到最新的值，这种现象也被称之为 不可重复读 。

4. 幻读（Phantom）

对于两个事务Session A、Session B, Session A 从一个表中 读取 了一个字段, 然后 Session B 在该表中 插入 了一些新的行。 之后, 如果 Session A 再次读取 同一个表, 就会多出几行。那就意味着发生了幻读。

Session A中的事务先根据条件 studentno > 0这个条件查询表student，得到了name列值为'张三'的记录；之后Session B中提交了一个 隐式事务 ，该事务向表student中插入了一条新记录；之后Session A中的事务再根据相同的条件 studentno > 0查询表student，得到的结果集中包含Session B中的事务新插入的那条记录，这种现象也被称之为 幻读 。我们把新插入的那些记录称之为 幻影记录 。

事务隔离级别

3.3 SQL中的四种隔离级别

上面介绍了几种并发事务执行过程中可能遇到的一些问题，这些问题有轻重缓急之分，我们给这些问题按照严重性来排一下序：

```
脏写 > 脏读 > 不可重复读 > 幻读
```

我们愿意舍弃一部分隔离性来换取一部分性能在这里就体现在：设立一些隔离级别，隔离级别越低，并发问题发生的就越多。 SQL标准 中设立了4个 隔离级别 ：

- `READ UNCOMMITTED` ：读未提交，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。不能避免脏读、不可重复读、幻读。
- `READ COMMITTED` ：读已提交，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。这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。可以避免脏读，但不可重复读、幻读问题仍然存在。
- `REPEATABLE READ` ：可重复读，事务A在读到一条数据之后，此时事务B对该数据进行了修改并提交，那么事务A再读该数据，读到的还是原来的内容。可以避免脏读、不可重复读，但幻读问题仍然存在。这是MySQL的默认隔离级别。
- `SERIALIZABLE` ：可串行化，确保事务可以从一个表中读取相同的行。在这个事务持续期间，禁止其他事务对该表执行插入、更新和删除操作。所有的并发问题都可以避免，但性能十分低下。能避免脏读、不可重复读和幻读。

SQL标准 中规定，针对不同的隔离级别，并发事务可以发生不同严重程度的问题，具体情况如下：

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCONMITED	Yes	Yes	Yes	No
READ COMMITED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

脏写 怎么没涉及到？因为脏写这个问题太严重了，不论是哪种隔离级别，都不允许脏写的情况发生。

不同的隔离级别有不同的现象，并有不同的锁和并发机制，隔离级别越高，数据库的并发性能就越差，4种事务隔离级别与并发性能的关系如下：

14，MySQL事务日志

而事务的原子性、一致性和持久性由事务的 redo 日志和undo 日志来保证。

- **REDO LOG 称为 重做日志** ，提供再写入操作，恢复提交事务修改的页操作，用来保证事务的持久性。
- **UNDO LOG 称为 回滚日志** ，回滚行记录到某个特定版本，用来保证事务的**原子性**

- 事务的状态：
  - 活跃的：一个正在执行的事务
  - 部分提交的：指的是也执行完最后一条语句，在内存中修改但未刷新到磁盘
  - 失败的：指的是处于活跃或者部分提交状态的事务出现了意外

- 终止：失败的事务回滚到初始状态
- 提交：部分提交的事务刷新到磁盘中。

1，REDO日志

- 在处理数据库数据时，构建缓冲池，总是先修改内存中的数据。然后再更新到数据库。缓冲池可以帮助我们消除CPU和磁盘之间的鸿沟。
- 由于事务的持久性，修改内存和修改磁盘有时间间隔，也不会每次操作都修改磁盘，操作时间慢。所以使用了redo日志。
- redo日志记录事务处理的所有操作，并保存在磁盘中。内存每1s进行刷盘。若服务器宕机，则按redo日志记录操作进行持久化操作。

好处：

- redo日志降低了刷盘频率
- redo日志占用的空间非常小
  内容是存储表空间，页号，偏移量以及需要更新的值，所需的空间小，刷盘快

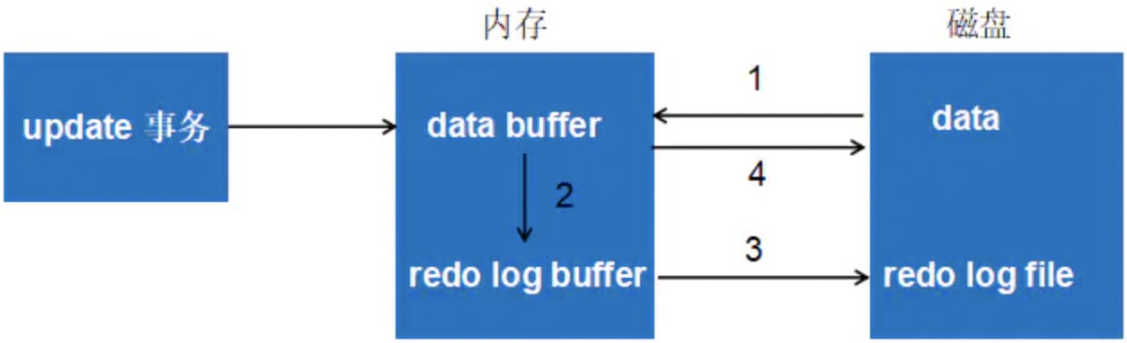
特点：

- redo日志是顺序写入磁盘的
- **事务执行过程中，redo log不断记录**

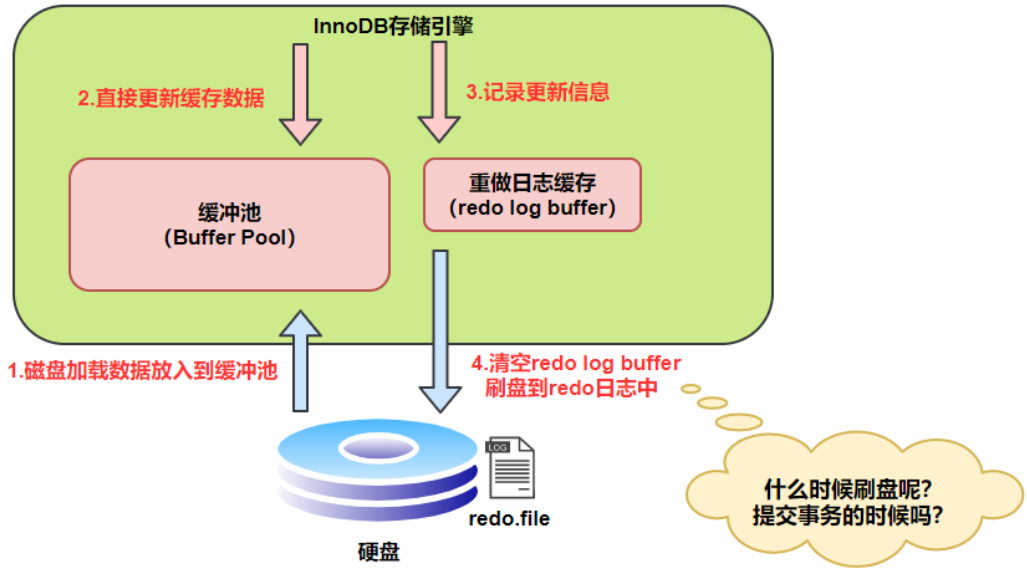
  bin log是在事务提交后，在一次性的写入日志

redo 日志的组成：有内存和文件两部分组成，redo log buffer和redo log file；

redo Log整体流程：



1. 更新事务提交后，首先检查内存中是否有对应的数据，没有则在磁盘中读取
2. 加载到内存中后，对数据执行事务。执行事务的同时，写入redo log 缓冲中。
3. 执行完成后，将redo log缓冲中的内容写入redo log file。
4. 定期进行刷盘操作，将数据缓冲写入到磁盘
5. 若服务器宕机，则通过redo log file进行持久化操作



redo log的刷盘策略：

缓存刷新到redo log 文件的三种情况

- 主线程每秒将redo log buffer刷盘到 redo log file
- 每次事务提交将进行刷盘
- redo log buffer神域空间小于1/2时，进行刷盘

redo log buffer刷盘到 redo log file的过程并不是真正刷到磁盘，而是写入到文件系统缓存（page cache）中去（这是现代操作系统为了提高文件写入效率做的一个优化），真正的写入会交给系统自己来决定。

针对这种情况，InnoDB通过修改innodb_flush_log_at_trx_commit参数可以更改刷盘策略。即如何将 redo log buffer 中的日志刷新到 redo log file 中

- 设置为0： 事务提交不刷盘，使用系统刷盘（**系统默认master thread每隔1s进行一次重做日志的同步**）
- 设置为1： 每次事务提交就刷盘 **（ 默认值 ）**
- 设置为2： 表示每次事务提交时都只把 redo log buffer 内容写入 page cache，不进行同步。由os自己决定什么时候同步到磁盘文件。

redo log file写入策略：

日志文件的大小是有限，因此可以采用循环使用的方式，当数据已满时，就会将一部分已经更新的文件清空。

2，undo 日志

undo日志是事务原子性的保证，即在事务中，执行失败，使用undo日志进行回滚，恢复到事务执行前的状态。

- 对记录做出改动，则会将回滚时需要的数据记录下来。（**增，删，改回记录日志，查不会记录**）
- undo日志也会产生redo log。因为回滚日志也需要持久化的保护

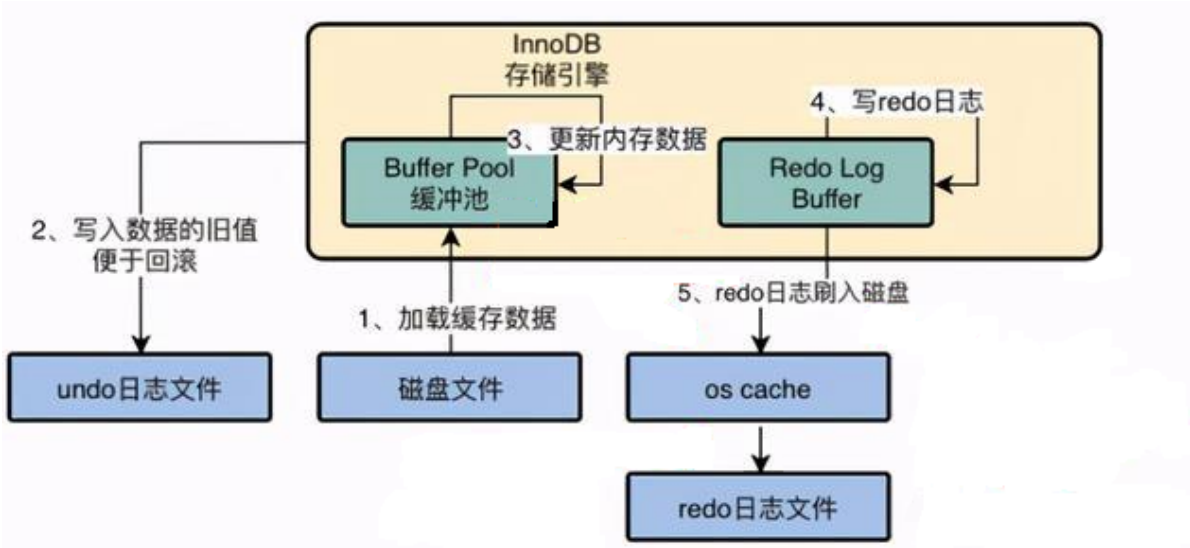
作用：

- 用于回滚，保证事务的一致性
- MVCC，MySQL中可以用于多版本并发控制

undo log 重用

因为操作多，undo log生成多。另外mysql操作单位是页，16K。undo log可以判断是否可以重用。小于3/4，可以重用。重用是在当前undo 页后面记录当前的数据

- uodo log的分类和删除：
  - insert操作：对于insert undo log来说，因为数据仅对当前事务可见，提交后可以直接删除
  - update操作：对于update undo log来说，当前版本中的数据在MVCC中可以会供其他读操作使用，因此不能删除



redo log是物理日志，记录的是数据页的物理变化，undo log不是redo log的逆过程。

3，CheckPoint技术

CheckPoint技术：将缓存池中的脏页刷会到磁盘。有两种CheckPoint技术

- Sharp CheckPoint：数据库关闭时，将所有的脏页刷回到磁盘
- Fuzzy CheckPoint：只刷新部分脏页，并非将所有脏页刷会磁盘。

起因：

在执行数据库修改操作时，宕机，通过redo log日志进行恢复。redo log日志大小有限制，数据库不能完全依赖刷盘后的redo log日志来恢复从数据库启动时的所有数据。需要CheckPoint指针进行标记，在CheckPoint指针标记点后面的redo log日志进行恢复。**若是redo log日志被覆盖重写时，发现redo log日志还需要使用，此时通过CheckPoint技术进行刷盘**

Fuzzy CheckPoint使用的情况

- 缓存池不够用，LRU算法清退最近最少使用的页，如果是脏页，则执行CheckPoint技术刷盘
- redo log日志被覆盖重写时，发现redo log日志还需要使用，此时通过CheckPoint技术进行刷盘
- 主线程每秒会将缓存池的脏页按比例刷回到磁盘，异步操作，不影响主线程
- Mysql 5.6版本后，设置LRU队列可用页的数量，默认值1024。若低于这个数字，则执行Check Point。
- Dirty Page 脏页数量太多，默认是超过75%，则进行刷页

15，锁

1，并发问题：

读-写 或 写-读 ，即一个事务进行读取操作，另一个进行改动操作。这种情况下可能发生 脏读 、 不可重复读 、 幻读 的问题。

- 方案一：读操作利用多版本并发控制 （ MVCC ） ，写操作进行 加锁 。
- 方案二：读、写操作都采用 加锁 的方式。

采用 MVCC 方式的话， 读-写 操作彼此并不冲突， 性能更高 。
采用 加锁 方式的话， 读-写 操作彼此需要 排队执行 ，影响性能。

- 锁的分类
    - 数据类型的类型：写锁（排他锁），读锁（共享锁）
    - 操作的粒度：表锁，行锁和页锁
    - 对其他锁的态度：共享锁和排他锁

共享锁，排他锁

- 共享锁 s锁： 不会影响数据读取，可以多线程读
- 排他锁 x锁： 只能有一个事务操作，防止其他用户的读写

需要注意的是对于 InnoDB 引擎来说，读锁和写锁可以加在表上，也可以加在行上。

注意：**读操作可以加s锁也可以加x锁，写操作只能加x锁**

需要注意的是对于 InnoDB 引擎来说，读锁和写锁可以加在表上，也可以加在行上。

表级锁，行锁

- LOCK TABLES t READ： InnoDB存储引擎会对表 t 加表级别的 S锁 。
- LOCK TABLES t WRITE： InnoDB存储引擎会对表 t 加表级别的 X锁 。

加行锁 ：排他锁

使用锁的条件

```
创建行锁条件：

1、表中创建索引， select 。。。 where  字段（必须是索引）  不然行锁就无效。

2、必须要有事务，这样才是 行锁（排他锁）

3、在select  语句后面 加 上   FOR UPDATE;
```

InnoDB可以使用行级锁，表锁，MyISAM只能使用表锁。

InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

- 表锁
  - 表级别的x，s锁：InnoDB一般使用行锁不使用表锁，
    使用场景：InnoDB在修改表结构的时候会加上表级锁
    MyISAM使用的只有表级的x，s锁
    - s锁：在加锁期间，所有人可读，所有人都不能写。加锁期间也不能操作（增删改查）其他表。只有解锁后才可以
    - x锁：在加锁期间，该用户可读，可写。其他人读写该表都不可。而不能操作其他表
  - **意向锁（表级别） 行锁和表锁共存**
    - 使用场景：用户在表中某行加入了行锁，会在更加的一级空间（页锁或者表锁）添加意向锁，通知其他操作，已经上过意向锁。不必再遍历查询是否表中有锁。因此，实现了行锁和表锁的共存。
    - **添加行锁后，存储引擎自动添加意向锁**，用户无法操作。分为意向共享锁 IS和意向排他锁 IX
    - 意向锁与表级的xs锁是阻塞的，is和s不会。意向锁与意向锁之间都是兼容的
  - 元数据锁（MDL锁）：对表的结构做出变更时，会有MDL写锁。对表做增删查改时加MDL读锁。防止修改表数据时，表结构发生变化。
- InnoDB的行锁
  - 记录锁：与表锁特性相同 共享锁，排他锁
  - 间隙锁：**只为了解决幻读问题，**禁止该行所在间隙插入数据。共享和独占锁作用相同。
    - 比如数据为 1,5。插入间隙锁可以是2，3，4都可以加间隙锁。开区间
  - 临键锁：锁住该条记录以及该行与上一行的间隙
    - 记录锁和间隙锁的结合。左开有闭
  - 插入意向锁：由于间隙锁，其他事务在该间隙插入数据，会等待。等待期间会在内存中生成一个插入意向锁，表明要在该行插入数据的意向。

乐观锁，悲观锁

- 悲观锁
  - 每次操作数据都会上锁，行锁表锁都是悲观锁。
  - **适用场景：**适合写操作多的场景
- 乐观锁
  - 乐观锁是通过程序来实现
  - 每次操作不会上锁。通过版本号机制，时间戳机制（CAS）
  - **适用场景：**适合读操作多的场景

16，MVCC

1，简介：

通过对多个版本的管理来处理数据库的并发问题。在一个读事务读取的数据正在被其他事务操作时，可以读取该数据的旧版本。

多版本并发控制 => MVCC。读写操作出现的事务问题（脏读，脏写，幻读，不可重复读），可以通过MVCC解决。

解决的隔离级别是读已提交和可重复读，与临键锁结合，可以解决幻读问题

2，快照读和当前读

快照读：读取数据，若数据正在被其他事务操作，则读取旧版本的数据。普通不加锁的读就是快照读

当前读：读取的数据一定是最新版本的数据。 **加锁和修改操作都是当前读，与MVCC无关**

3，MVCC实现原理

- MVCC的实现依赖于：**隐藏字段，Undo log，ReadView**
- **隐藏字段：　Undo log 对每一次的修改都会保存一下数据**
  - 聚簇索引中保存： 记录插入或更新该行的最后一个事务的事务 ID，和回滚指针（寻找undo log）
    - trx_id ： 每次事务对数据进行改动时，聚簇索引都会把该事务的事务id赋值给trx_id
    - roll_pointer ： 每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到 undo日志中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。
- **undo log：**是MySQL读取以前版本的数据来源。
- MVCC使用**快照读**来解决并发问题。**解决的隔离级别是读已提交和可重复读。**ReadView中包含
  - 创建该ReadView的事务ID
  - 生成ReadView时其他未提交的事务ID列表或者说活跃的事务ID列表，或者说集合
  - 活跃事务的最小ID
  - 所有ReadView（包括现在活跃的和已提交的）的最大事务id，递增值
- MVCC版本控制的基本原则：

  （根据这个ReadView，判断哪个版本是可见的）
  - 若当前被访问版本的事务id与ReadView的事务ID相同，则当前版本可以访问
  - 若当前访问版本的事务ID小于ReadView活跃事务的最小ID，则当前版本可以访问　（表明该版本的事务早于我们事务，而且已经提交）
  - 若当前访问版本的事务ID大于所有ReadView最大事务ID，则不能访问　（说明该版本晚于我们的事务，时间上不允许读取的）
  - 当前访问版本的事务ID大于ReadView活跃事务的最小ID，小于所有ReadView最大事务ID。需要二次判断
    - 当前访问版本的事务ID是否在活跃的事务ID 列表（即是否已经提交）
    - 在列表，则不能访问，不在则可以访问

回滚指针指向该条记录的undo log日志，该日志记录的该行数据是一个版本链，倒序遍历该链来找到匹配的数据记录。

ReadView与每一个Undo Log的版本进行匹配。（遍历的事务Undo log中的各个版本）

4，MVCC整体操作流程

查询一条记录如何通过MVCC获取

1. 首先获取事务自己的ID
2. 获取ReadView
3. 若查询到数据然后与ReadView中的事务ID比较
4. 不符合规则，则从Undo log中获取历史版本【即该行数据的版本链】，挨个遍历
5. 最后返回符合规则的数据

MVCC只在这两个隔离级别起作用　--->可重复和不可重复读：

- **不可重复读（读已提交）：**同一次事务，每次select都会产生一个ReadView，会生成不同活跃列表等信息。所以可能会读取到不同的数据
- **可重复读：**在事务开始的时候生成一个readview供全部的select使用，因此是可以重复读
- **可解决幻读**：与临键锁结合，可以解决幻读问题

MVCC解决解决的问题：

不可重复读，幻读问题

17，常见日志

1，常见日志

- 通用查询日志：记录MySQL从启动到终止过程中的执行所有指令的日志
- **二进制日志：**记录所有更改数据的语句，用于主从复制，以及服务器遇到故障时的无损失恢复
- **中继日志：**主从复制中，从服务器存放二进制日志内容的一个中间件

2，二进制日志

binary log：它记录了数据库所有执行的DDL和DML等数据库内容进行修改的操作。

binary log场景：

- **数据恢复**
- 数据复制

3，bin log和redo log对比

- redo log是物理日志，记录的内容是数据做出什么修改。InnoDB存储引擎所特有的日志文件，只记录该引擎下的信息【redo log 日志记录的是每个页更改的物理情况】
- bin log是逻辑日志，记录内容是语句的原始逻辑 。会记录MySQL数据库有关的包括InnoDB，MyIsam，Heap等存储引擎的数据库。【bin log 日志记录的是一个事务的具体操作日志】
- 虽然都是属于持久化的保证，但是侧重点不同
    - redo log 让InnoDB存储引擎拥有崩溃恢复的能力，持久化数据库的数据
    - bin log 保证了主从复制的数据一致性
- redo log 是执行事务时就写入数据，bin log日志是事务一提交才写入日志

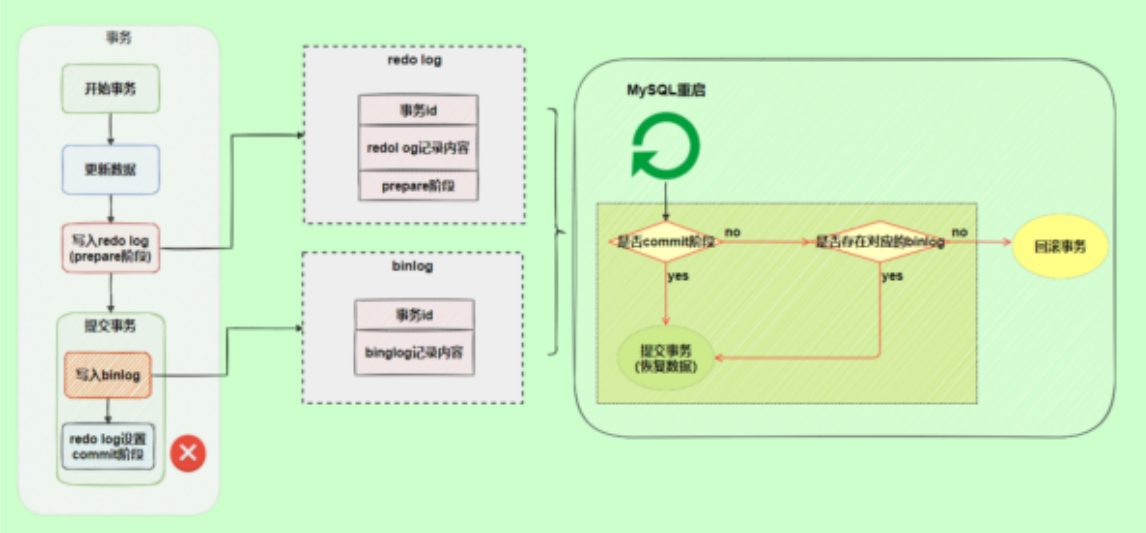
4，两阶段提交

redo log 是执行事务后就写入数据，bin log日志是事务一提交才写入日志。若事务提交后，出现异常，导致redo log日志写入完成，bin log日志未写入。导致从机数据未更新。

所以选择两阶段提交：

redo log日志写入分为两个阶段，

- 若在写入 bin log日志出现异常，则redo log没有第二阶段，所以会进行回滚。
- 若relo log设置commit阶段发生异常，不会回滚。因为发现bin log日志已经写入，不必回滚



binary log日志格式

- Statement
  - 记录每一条修改数据库的SQL语句。**默认的bin log 格式**
  - 优点：
    - bin log日志较小，不必记录每一行数据的变化，
    - 主从版本可以不一样，从服务器版本可以比主服务器版本高
  - 缺点：
    -
- Rows
  - 不记录指令，只记录数据库修改的内容
  - 优缺点：为数据库的恢复和复制带来可靠性，bin log 文件大
- MIXED模式
  - Statement和Row的结合，一般语句使用statment格式保存binlog。一些函数，使用row格式保存binlog。

中继日志

中继日志只在主从服务器架构的从服务器上存在。从服务器为了与主服务器保持一致，要从主服务器读取二进制日志的内容，并且把读取到的信息写入 本地的日志文件 中，这个从服务器本地的日志文件就叫中继日志 。然后，从服务器读取中继日志，并根据中继日志的内容对从服务器的数据进行更新，完成主从服务器的 数据同步 。

18，主从复制

主从复制是实现读写分离。读操作和写操作在不同服务器上。用户“读多写少”，所以分离操作，缓解数据库压力。

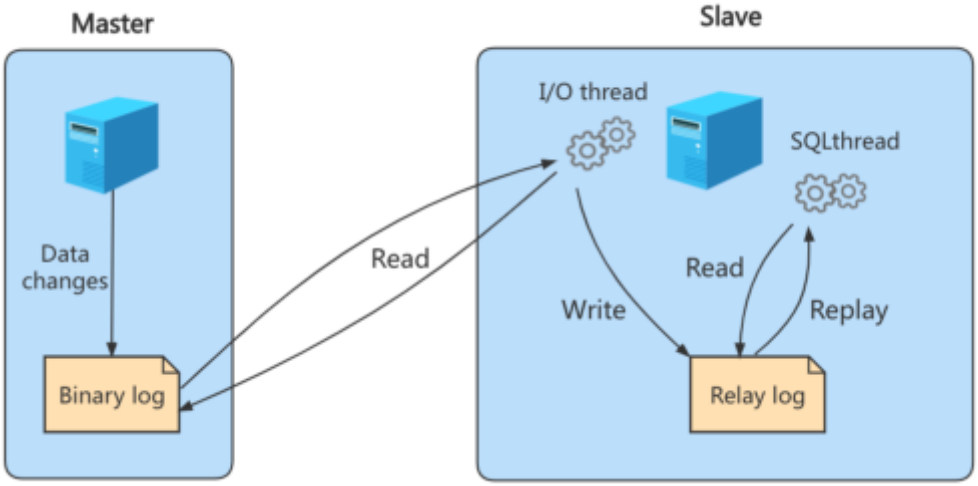
1，主从复制作用

- 提高数据库的吞吐量
- 读写分离
- 数据备份
- 具有高可用性

2，主从复制过程

主从复制会生成3个线程来操作，一个主库线程，两个从库线程

- 二进制日志线程：主机将数据库变更添加到bin log
- 从库I/O线程：读取主库的bin log日志，并写入到从机的中继日志 Relay log
- 从库SQL线程：读取中继日志内容并执行响应操作



复制的最大问题：延时

主从复制过程中，会有4次io的读写操作，势必慢

原则：

- 每个从机只能有一个master（主机）
- 每个slave只能有一个唯一的服务器ID
- 每个master可以有多个slave，任意多台从机

杂七杂八知识点

- varchar(n) n指定的不是字节长度，而是容纳长度，如：一个utf-8的汉字需要3个字节存储，那么varchar（n）可以最大存储n个汉字，即3*n个字节。utf-8的字母则是一个字节，存储的内容MySQL会进行分析，找出最少的存储方式。