**mysql**

* 高级
  + 索引
    - 概述
      * 定义
      * 白话说就是为了满足数据的高效查找而设计的一种数据结构
      * 分类
        + B+树索引

维度1

聚集索引

索引项的排序和对应数据的排序一致

非聚集索引

索引项的排序和对应的数据排序不一致

维度2

主索引

对主键建立的索引叫做主索引

辅助索引

对非主键建立的索引叫做辅助索引

维度3

唯一索引

表中的某一个字段是唯一的且对这个字段建立了索引

非唯一索引(普通索引)

表中的某个字段不是唯一的且对这个字段建立了索引

维度4

联合索引

对表中多个字段建立了索引

覆盖索引

对表中多个字段建立了索引且满足查找这个索引就能搜索出需要的数据

* + - * + 全文索引
        + 即fulltext,是提高全文搜索速度的一种索引结构
        + 哈希索引
        + InnoDB存储引擎支持的哈希索引是自适应的，InnoDB存储引擎会根据表的使用情况自动为表生成哈希索引，不能人为干预是否在一张表中生成哈希索引
    - 特点
      * 聚集索引
        + 唯一
        + 每张表只能拥有一个聚集索引,是对主键建立的索引
        + 数据页
        + 聚集索引的叶子节点称为数据页，因此在多数情况下，查询优化器倾向于采用聚集索引。因为聚集索引能够在B+树索引的叶子节点上直接找到数据。
        + 加快范围查询
        + 聚集索引能够特别快地访问针对范围值的查询。查询优化器能够快速发现某一段范围的数据页需要扫描
      * 辅助索引
        + 不是全部数据
        + 叶子节点并不包含行记录的全部数据,叶子节点除了包含键值以外,还包括相应行数据的聚集索引键
        + 可以建立多个
        + 可以建立多个辅助索引
        + 可能引起离散读
        + 可能引起离散读的问题,但是一般的数据库都通过实现预读（read ahead）技术来避免多次的离散读操作
      * 联合索引
        + 多列建立
        + 可以对多个列进行建立联合索引,加快搜索速度
        + 有序
        + 已经对第二个键值进行了排序处理。例如，在很多情况下应用程序都需要查询某个用户的购物情况，并按照时间进行排序，最后取出最近三次的购买记录，这时使用联合索引可以避免多一次的排序操作，因为索引本身在叶子节点已经排序了
      * 覆盖索引
        + 比较小
        + 不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以减少大量的IO操作
        + 统计优化
        + 对某些统计问题而言的,比如:SELECT COUNT(\*) FROM tablename;InnoDB存储引擎并不会选择通过查询聚集索引来进行统计。由于buy\_log表上还有辅助索引，而辅助索引远小于聚集索引，选择辅助索引可以减少IO操作
    - 使用
      * Cardinality
        + 高低选择性
        + 如果一个字段，它可取值的范围很小，称为低选择性
        + 如果某个字段的取值范围很广，几乎没有重复，即属于高选择性
        + 定义
        + 表示索引中不重复记录数量的预估值
        + 统计原理

统计方式

通过采样法来完成:

取得B+树索引中叶子节点的数量，记为A。

随机取得B+树索引中的8个叶子节点。统计每个页不同记录的个数，即为P1，P2，…，P8。

根据采样信息给出Cardinality的预估值：Cardinality=（P1+P2+…+P8）\*A/8。

更新策略

在insert和update满足一定条件时触发

表中1/16的数据已发生过变化。

表中发生变化的次数＞2 000 000 000

执行某些sql语句时

当执行SQL语句ANALYZE TABLE、SHOW TABLE STATUS、SHOW INDEX以及访问INFORMATION\_SCHEMA架构下的表TABLES和STATISTICS时会导致InnoDB存储引擎去重新计算索引的Cardinality值

* + - * 建立索引的依据
        + 索引字段具有高选择性
        + 从表中取出较少的数据
    - 索引失效
      * 模
      * 指的是模糊匹配,对于%开头的模糊匹配key,例如%key,索引会失效;但是形如key%,索引不会失效
      * 型
      * 当sql语句where后使用的key数据类型和表中数据类型不一致时,索引会失效。比如where key = 1，表中key是vchar类型，索引会失效.这种工作中比较容易犯,应当重点注意
      * 数
      * 指的是对索引的字段进行函数计算,比如形如where … avg(key)等，索引会失效
      * 空
      * 指的是索引的字段在表中可能存在空值,索引会失效
      * 运
      * 指的是对索引的字段进行相关的计算,例如加减等等,索引会失效
      * 最
      * 大名鼎鼎的联合索引最左匹配原则
      * 快
      * 当sql优化器估测不使用索引更快时
      * 其他
      * 比如范围查询 join 等也可能会导致索引失效
    - 存储引擎优化
      * Multi-Range Read(MRR)优化
        + 定义
        + MySQL5.6版本开始支持Multi-Range Read（MRR）优化。Multi-Range Read优化的目的就是为了减少磁盘的随机访问，并且将随机访问转化为较为顺序的数据访问，这对于IO-bound类型的SQL查询语句可带来性能极大的提升
        + 适用范围

range

ref

eq\_ref

* + - * + 优势

原理

使数据访问变得较为顺序

在查询辅助索引时，首先根据得到的查询结果，按照主键进行排序，并按照主键排序的顺序进行书签查找。

减少缓冲池中页被替换的次数

批量处理对键值的查询操作

算法过程

存储辅助索引的键值并排序

将查询得到的辅助索引键值存放于一个缓存中，这时缓存中的数据是根据辅助索引键值排序的

将缓存中的键值根据RowID进行排序

根据RowID的排序顺序来访问实际的数据文件

* + - * + 举例

使数据访问变得较为顺序

SELECT \* FROM salaries WHERE salary＞10000 AND salary＜40000;

先缓存满足条件的辅助索引键值，然后根据rowid进行排序，接着较为顺序的访问数据文件

将范围查询拆分为键值对

SELECT \* FROM t WHERE key\_part1＞=1000 AND key\_part1＜2000 AND key\_part2=1000;

表t有（key\_part1，key\_part2）的联合索引，优化器会先将查询条件进行拆分，然后再进行数据查询同时会过滤不满足的数据。就上述查询语句而言，优化器会将查询条件拆分为（1000，1000），（1001，1000），（1002，1000），…，（1999，1000），最后再根据这些拆分出的条件进行数据的查询。

* + - * Index Condition Pushdown(ICP)优化
        + 定义
        + MySQL数据库会在取出索引的同时，判断是否可以进行WHERE条件的过滤，也就是将WHERE的部分过滤操作放在了存储引擎层。在某些查询下，可以大大减少上层SQL层对记录的索取（fetch），从而提高数据库的整体性能
        + 优点
        + 在取出索引的同时，判断是否可以进行WHERE条件的过滤，也就是将WHERE的部分过滤操作放在了存储引擎层。在某些查询下，可以大大减少上层SQL层对记录的索取（fetch），从而提高数据库的整体性能。
        + 举例
        + explain中Extra字段带有Using index condition 就是表示ICP优化
    - 全文索引
      * 定义
      * 是将存储于数据库中的整本书或整篇文章中的任意内容信息查找出来的技术
      * 使用限制
        + 每张表只能有一个全文检索的索引
        + 由多列组合而成的全文检索的索引列必须使用相同的字符集与排序规则
        + 支持没有单词界定符(delimiter)的语言，如中文、日语、韩语等
  + 锁
    - 定义
    - 协调多进程或者多线程并发访问共享资源的一种机制
    - 模型
      * latch(闩锁(轻量级的锁))
        + 特点
        + 锁定的时间必须非常短
        + 分类

互斥量

读写锁

* + - * + 操作对象
        + 共享资源
      * lock(锁)
        + 特点
        + commit和rollback后释放
        + 有死锁检测机制
        + 操作对象
        + 事务
    - 类型
      * 行级锁
        + 共享锁
        + 排它锁
      * 表级锁
        + 共享锁
        + 排它锁
        + 意向共享锁

定义

事务有意向对表中的某些行加共享锁

特点

行级共享锁前提

如若想获取表中某些行的共享锁,必须先获取表的意向共享锁

与表级共享锁兼容

与表级排它锁互斥

不与行级共享锁互斥

不与行级排它锁互斥

与意向排它锁兼容

* + - * + 意向排他锁

定义

事务有意向对表中的某些行加排他锁

特点

行级排它锁前提

如若想获取表中某些行的排他锁,必须先获取表的意向排它锁

与表级共享锁互斥

与表级排它锁互斥

不与行级共享锁互斥

不与行级排它锁互斥

与意向共享锁兼容

* + - 兼容性
    - 按照正常的共享排它锁逻辑理解即可
    - 一次性非锁定读
      * 定义
      * 是指InnoDB存储引擎通过行多版本控制（multi versioning）的方式来读取当前执行时间数据库中行的数据。如果读取的行正在执行DELETE或UPDATE操作，这时读取操作不会因此去等待行上锁的释放。相反地，InnoDB存储引擎会去读取行的一个快照数据
      * 读取原理
      * 通过多版本并发控制(MVCC)实现
      * 快照定义
        + READ COMMITTED
        + 非一致性读总是读取被锁定行的最新一份快照数据
        + REPEATABLE READ
        + 非一致性读总是读取事务开始时的行数据版本
      * 优点
      * 极大的提高了数据库的并发性
    - 一次性锁定读
      * 定义
      * 在某些情况下，用户需要显式地对数据库读取操作进行加锁以保证数据逻辑的一致性
      * 分类
        + SELECT … FOR UPDATE
        + SELECT … LOCK IN SHARE MODE
        + 注意事项:以上2种模式都需要在一个事务中
    - 行锁算法
      * Record Lock
      * 单个行记录上的锁
      * Gap Lock
      * 间隙锁,锁定一个范围,但是不包括记录本身
        + 关闭Gap Lock
        + 1. 将事务的隔离级别设置为READ COMMITTED
        + 2. 将参数innodb\\_locks\\_unsafe\\_for\\_binlog设置为1
      * Next-Key Lock
      * Record Lock + Gap Lock.锁定一个范围,并且锁定记录本身
        + 目的
        + 为了解决REPEATABLE READ级别下幻读问题
        + 注意事项
        + 当查询的索引含有唯一属性时，InnoDB存储引擎会对Next-Key Lock进行优化，将其降级为Record Lock，即仅锁住索引本身，而不是范围
    - 锁问题
      * 脏读
        + 定义
        + 在不同的事务下，当前事务可以读到另外事务未提交的数据，简单来说就是可以读到脏数据
        + 产生条件

需要事务的隔离级别为READ UNCOMMITTED

别的事务修改数据但是未提交

* + - * + 解决办法
        + 事务的最少隔离级别为READ-COMMITTED
        + 适用特例
        + 例如replication环境中的slave节点，并且在该slave上的查询并不需要特别精确的返回值。
      * 不可重复读
        + 定义

维基百科

在一个事务内两次读到的同一(行)数据是不一样的情况

mysql官方文档

将不可重复读的问题定义为Phantom Problem，即幻读问题。即在一个事务内两次读到的数据是不一样的情况(包括update、insert和delete)

* + - * + 产生条件

事务的隔离级别最高为READ-COMMITTED

别的事务对同一(行)数据执行update并且已提交

* + - * + 解决办法
        + 事务隔离级别最少是READ REPEATABLE，采用一致性非锁定读和Next-Key Lock算法避免不可重复读
      * 幻读
        + 定义
        + 维基百科中定义:当在事务过程中，另一事务向正在读取的记录添加或删除新行时，将发生幻像读取(读取的数据不一样)
        + 产生条件

事务的隔离级别最高为REPEATABLE-READ

别的事务进行insert或者delete操作，并且已提交

* + - * + 解决办法

将事务的隔离级别最少提升至REPEATABLE-READ

采用Next-Key Lock算法

mysql的REPEATABLE-READ级别下采用了Next-Key Lock算法,所以mysql的该级别下解决了幻读问题

* + - * 丢失更新
        + 定义
        + 简单来说其就是一个事务的更新操作会被另一个事务的更新操作所覆盖，从而导致数据的不一致。
        + 注意:理解上面的定义关键在于,mysql数据库Innodb默认的数据库隔离级别是:REPEATABLE-READ,T1 T2事务采取一致性非锁定读, 读取的是事务开始时的行数据版本.
        + 产生条件

需要事务的隔离级别最高为REPEATABLE-READ

采用一致性非锁定读的方式读取事务开始时的数据

* + - * + 解决办法

加上排它锁,将T1 和 T2事务串行化

将数据库隔离级提升至SERIALIZABLE

* + - 死锁
      * 定义
      * 是指两个或两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象。若无外力作用，事务都将无法推进下去
      * msyql中死锁分类
        + AB-BA死锁
        + A事务在等待B事务，B事务在等待A事务.
        + 事务持有X锁又请求S锁
      * 解决方式
        + 等待超时
        + 即当两个事务互相等待时，当一个等待时间超过设置的某一阈值时，其中一个事务进行回滚，另一个等待的事务就能继续进行
        + 有2个字段可以设置:
        + innodb\\_lock\\_wait\\_timeout
        + innodb\\_rollback\\_on\\_timeout
        + wait-for graph(等待图)

特点

一种更为主动的死锁检测方式

原理

wait-for graph要求数据库保存以下两种信息：

锁的信息链表

事务等待链表

通过上述链表可以构造出一张图，而在这个图中若存在回路，就代表存在死锁，因此资源间相互发生等待。

* + 事务
    - 定义
    - 是访问并更新数据库各种数据项的一个程序执行单元,要么都做修改 要么都不做
    - 事务ACID特性
      * 原子性(atomicity)
      * 简单来说 要么都做 要么都不做
      * 一致性(Consistency)
      * 一致性是指事务将数据库从一种状态转变为下一种一致的状态,在事务开始之前和事务结束之后,数据库的完整性约束没有被破坏
      * 隔离性(Isolation)
      * 每个事务的操作对象都能够分离，即事务提交前对其他事务都不可见
      * 持久性(duraibility)
      * 事务一旦提交,其结果就是永久性的，即使数据库宕机 数据也能恢复
    - 事务实现原理
      * 锁事务的隔离性
      * redo log实现事务的原子性和持久性
      * undo log实现事务的一致性
    - 事务统计
      * QPS(Question Per Second)：每秒请求数
      * TPS(Transaction Per Second)：每秒事务处理能力
      * 统计方法
        + (com\_commit + com\_rollback)/time
        + 利用这种方法进行计算的前提是：所有的事务必须都是显式提交的，如果存在隐式地提交和回滚（默认autocommit=1），不会计算到com\\_commit和com\\_rollback变量中
        + handler\_commit和handler\_rollback
        + 在MySQL 5.1中可以很好地用来统计InnoDB存储引擎显式和隐式的事务提交操作，但是在InnoDB Plugin中这两个参数的表现有些“怪异”，并不能很好地统计事务的次数
    - 事务隔离级别
      * READ UNCOMMITTED
      * READ COMMITTED
      * REPEATABLE READ
      * SERIALIZABLE
    - 分布式事务
      * 条件
        + InnoDB事务隔离级别必须是SERIALIZABLE
        + 参与在全局事务中的每个节点都支持XA事务
      * XA事务组成
        + 资源管理器
        + 通常就是各种数据库
        + 事务管理器
        + 协调参与全局事务中的各个事务。需要和参与全局事务的所有资源管理器进行通信
        + 应用程序
        + 定义事务的边界，指定全局事务中的操作
      * 两阶段提交
        + 第一阶段
        + 所有参与全局事务的节点都开始准备（PREPARE），告诉事务管理器它们准备好提交了
        + 第二阶段
        + 事务管理器告诉资源管理器执行ROLLBACK还是COMMIT。如果任何一个节点显示不能提交
    - 不好的事务习惯
      * 在循环中提交事务
      * 使用自动提交
      * 使用自动回滚
      * 使用长事务
  + 备份与恢复
    - 备份方式分类
      * 按备份方式
        + 冷备
        + 是指备份操作是在数据库停止的情况下，这种备份最为简单，一般只需要复制相关的数据库物理文件即可
        + 温备
        + 是在数据库运行中进行的，但是会对当前数据库的操作有所影响，如加一个全局读锁以保证备份数据的一致性
        + 热备
        + 是指数据库运行中直接备份，对正在运行的数据库操作没有任何的影响
      * 按备份文件的内容
        + 逻辑备份
        + 是指备份出的文件内容是可读的，一般是文本文件。内容一般是由一条条SQL语句，或者是表内实际数据组成
        + 裸文件备份
        + 是指复制数据库的物理文件，既可以是在数据库运行中的复制（如ibbackup、xtrabackup这类工具），也可以是在数据库停止运行时直接的数据文件复制
      * 按备份数据库的内容
        + 完全备份
        + 是指对数据库进行一个完整的备份
        + 增量备份
        + 是指在上次完全备份的基础上，对于更改的数据进行备份
        + 日志备份
        + 是指对MySQL数据库二进制日志的备份，通过对一个完全备份进行二进制日志的重做（replay）来完成数据库的point-in-time的恢复工作
    - 备份方式
      * 冷备
        + 备份方式
        + 只需要备份MySQL数据库的frm文件，共享表空间文件，独立表空间文件（\*.ibd），重做日志文件。另外建议定期备份MySQL数据库的配置文件my.cnf，这样有利于恢复的操作
        + 优点
        + 备份简单，只要复制相关文件即可。
        + 备份文件易于在不同操作系统，不同MySQL版本上进行恢复。
        + 恢复相当简单，只需要把文件恢复到指定位置即可。
        + 恢复速度快，不需要执行任何SQL语句，也不需要重建索引。
        + 缺点
        + InnoDB存储引擎冷备的文件通常比逻辑文件大很多，因为表空间中存放着很多其他的数据，如undo段，插入缓冲等信息。
        + 冷备也不总是可以轻易地跨平台。操作系统、MySQL的版本、文件大小写敏感和浮点数格式都会成为问题。
      * 热备
        + ibbackup
        + xtrabackup
      * 逻辑备份与恢复
        + 备份

mysqldump

SELECT … INTO OUTFILE

* + - * + 恢复

mysql

示例:shell > mysql -uroot -p < dump.sql

source

示例:mysql> source ~/test/dump.sql;

LOAD DATA INFILE

mysqlimport

* + - * 二进制日志备份与恢复
      * 快照备份
      * 复制 + 快照备份架构
  + 性能调优
    - CPU选择
    - 1.选择64位多核CPU
    - 2.增大读写线程个数
    - 3.OLAP属于CPU密集型
    - 内存选择
    - 提前预估活跃的数据量的大小,尽量选择大的内存
    - 硬盘选择
      * 传统机械VS固态
      * 传统机械硬盘顺序访问远高于随机访问
      * 固态硬盘读写是非对称的，读取速度大于写入速度
      * 调优方式
        + 考虑innodb\_io\_capacity参数
        + innodb\\_io\\_capacity:用来当刷新脏数据时，控制MySQL每秒执行的写IO量。
        + 如果是固态硬盘，考虑适当增大innodb\\_io\\_capacity,充分利用固态硬盘带来的高IOPS特性
        + 考虑关闭邻接页的刷新
        + 工作原理:当刷新一个脏页时，InnoDB存储引擎会检测该页所在区（extent）的所有页，如果是脏页，那么一起进行刷新。
        + 优点:通过AIO可以将多个IO写入操作合并为一个IO操作,故该工作机制在传统机械磁盘下有着显著的优势。
        + 传统机械键盘要开启该特性，固态硬盘可以选择关闭邻接页的刷新
        + OLTP选择固态
    - 合理设置RAID
      * RAID
        + 定义
        + RAID（Redundant Array of Independent Disks，独立磁盘冗余数组）的基本思想就是把多个相对便宜的硬盘组合起来，成为一个磁盘数组，使性能达到甚至超过一个价格昂贵、容量巨大的硬盘。
        + 作用
        + 增强数据集成度
        + 增强容错功能
        + 增加处理量或容量
      * RAID Write Back
        + 定义
        + RAID Write Back功能是指RAID控制器能够将写入的数据放入自身的缓存中，并把它们安排到后面再执行
        + 优点
        + 不用等待物理磁盘实际写入的完成，因此写入变得更快了。对于数据库来说，这显得十分重要。
      * 调优方式
        + 选择RAID10
        + 优点:读取与写入速度快,有镜像备份,可靠性高
        + 缺点:当一个磁盘失效时，性能可能会受到很大的影响，因为条带（strip）会成为瓶颈。
        + 视情况开启RAID Write Back功能
    - 操作系统选择
    - 使用64位的操作系统，并且使用64位mysql
    - 文件系统选择
    - 选择适合数据库的文件系统,比如支持快照等等
    - 基准测试工具的选择
      * sysbench
      * sysbench是一个模块化的、跨平台的多线程基准测试工具，主要用于测试各种不同系统参数下的数据库负载情况。
      * tpcc-mysql
      * 用来测试典型的复杂OLTP（在线事务处理）系统的性能
  + 集群
* 基础
  + mysql安装
  + #### sudo apt install mysql-server
  + #### sudo apt install mysql-client
  + InnoDB存储引擎
    - 概述
      * 后台线程
      * 后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最近的数据。此外将已修改的数据文件刷新到磁盘文件，同时保证在数据库发生异常的情况下InnoDB能恢复到正常运行状态。
        + Master Thread
        + Master Thread是一个非常核心的后台线程，主要负责将缓冲池中的数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新、合并插入缓冲（INSERT BUFFER）、UNDO页的回收等
        + IO Thread
        + 在InnoDB存储引擎中大量使用了AIO（Async IO）来处理写IO请求，这样可以极大提高数据库的性能。而IO Thread的工作主要是负责这些IO请求的回调（call back）处理
        + Purge Thread
        + 事务被提交后，其所使用的undolog可能不再需要，因此需要PurgeThread来回收已经使用并分配的undo页
        + Page Cleaner Thread
        + Page Cleaner Thread是在InnoDB 1.2.x版本中引入的。其作用是将之前版本中脏页的刷新操作都放入到单独的线程中来完成。而其目的是为了减轻原Master Thread的工作及对于用户查询线程的阻塞，进一步提高InnoDB存储引擎的性能
      * 内存池
        + 缓冲池
        + 由于CPU速度与磁盘速度之间的鸿沟，基于磁盘的数据库系统通常使用缓冲池技术来提高数据库的整体性能

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

* + - * + 重做日志缓冲
        + InnoDB存储引擎首先将重做日志信息先放入到这个缓冲区，然后按一定频率将其刷新到重做日志文件
        + 额外的缓冲池
        + 在InnoDB存储引擎中，对内存的管理是通过一种称为内存堆（heap）的方式进行的。在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。例如，分配了缓冲池（innodb\_buffer\_pool），但是每个缓冲池中的帧缓冲（frame buffer）还有对应的缓冲控制对象（buffer control block），这些对象记录了一些诸如LRU、锁、等待等信息，而这个对象的内存需要从额外内存池中申请。因此，在申请了很大的InnoDB缓冲池时，也应考虑相应地增加这个值。
    - 内存管理策略
      * LRU List管理
        + 改进版LRU策略

midpoint

在LRU列表中引入midpoint点(位于LRU首5/8处,可配置),把midpoint之后的列表称为old列表，之前的列表称为new列表。

innodb\_old\_blocks\_time

引入innodb\_old\_blocks\_time，用于表示页读取到mid位置后需要等待多久才会被加入到LRU列表的热端

* + - * + 优势

避免非活跃数据将活跃数据挤出缓冲池

对于索引或数据的扫描操作，并不是活跃的热点数据。如果页被放入LRU列表的首部，那么非常可能将所需要的热点数据页从LRU列表中移除，而在下一次需要读取该页时，InnoDB存储引擎需要再次访问磁盘

通过引入innodb\_old\_blocks\_time,尽可能降低使LRU列表中热点数据被刷出的可能

对于索引或数据的扫描操作，并不是活跃的热点数据。尽可能降低使热点数据被刷出的可能

* + - * Free List管理
        + 概述
        + LRU列表用来管理已经读取的页，但当数据库刚启动时，LRU列表是空的，即没有任何的页。这时页都存放在Free列表中。当需要从缓冲池中分页时，首先从Free列表中查找是否有可用的空闲页，若有则将该页从Free列表中删除，放入到LRU列表中。否则，根据LRU算法，淘汰LRU列表末尾的页，将该内存空间分配给新的页。当页从LRU列表的old部分加入到new部分时，称此时发生的操作为page made young，而因为innodb\_old\_blocks\_time的设置而导致页没有从old部分移动到new部分的操作称为page not made young
      * Flush List管理
        + 概述
        + 在LRU列表中的页被修改后，称该页为脏页（dirty page），即缓冲池中的页和磁盘上的页的数据产生了不一致。这时数据库会通过CHECKPOINT机制将脏页刷新回磁盘，而Flush列表中的页即为脏页列表。
    - 关键特性
      * 插入缓冲
        + Insert Buff

定义

对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个Insert Buffer对象中，

优劣势

优势

对于非聚集索引页不在缓冲池中先不插入,而是放入insert buff中,然后再以一定的频率和情况进行Insert Buffer和辅助索引页子节点的merge（合并）操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索引插入的性能。

劣势

Insert Buffer中如果有大量数据,若此时MySQL数据库发生了宕机，这时势必有大量的Insert Buffer并没有合并到实际的非聚集索引中去。因此这时恢复可能需要很长的时间，在极端情况下甚至需要几个小时

* + - * + Change Buff

定义

InnoDB从1.0.x版本开始引入了Change Buffer，可将其视为Insert Buffer的升级。从这个版本开始，InnoDB存储引擎可以对DML操作——INSERT、DELETE、UPDATE都进行缓冲，他们分别是：Insert Buffer、Delete Buffer、Purge buffer。

使用条件

1.索引是辅助索引

2.索引不是唯一的

开启

innodb\_change\_buffering

该参数可选的值为：inserts、deletes、purges、changes、all、none。inserts、deletes、purges就是前面讨论过的三种情况。changes表示启用inserts和deletes，all表示启用所有，none表示都不启用。该参数默认值为all。

* + - * + Merge Insert Buffer

辅助索引页被读取到缓冲池中

如果辅助索引页被读取到缓冲池中时，通过Insert Buffer Bitmap页确认该辅助索引页是否有记录存放于Insert Buffer B+树中,如果有,则将Insert Buffer B+树中该页的记录插入到该辅助索引页中。可以看到对该页多次的记录操作通过一次操作合并到了原有的辅助索引页中，因此性能会有大幅提高。

该辅助索引页已剩余存储空间少于1/32时

Insert Buffer Bitmap页追踪到该辅助索引页插入记录后可用空间会小于1/32页，则会强制进行一个合并操作，即强制读取辅助索引页，将Insert Buffer B+树中该页的记录及待插入的记录插入到辅助索引页中。

Master Thread机制

在Master Thread线程中每秒或每10秒会进行一次Merge Insert Buffer的操作。

Master Thread Merge 算法

srv\_innodb\_io\_capactiy

根据srv\_innodb\_io\_capactiy的百分比来决定真正要合并多少个辅助索引页

算法思想

随机地选择Insert Buffer B+树的一个页，读取该页中的space及之后所需要数量的页。该算法在复杂情况下应有更好的公平性。

* + - * 两次写
        + 定义
        + InnoDB存储引擎中有页的副本doublewrite Buffer,每次写入数据时，先写入共享表空间的物理磁盘上(顺序写),然后在写入页的各个表空间文件中(ibd 离散写)
        + 特性
        + 提升数据页的可靠性。
        + 当发生数据库宕机时，可能InnoDB存储引擎正在写入某个页到表中，而这个页只写了一部分，比如16KB的页，只写了前4KB，之后就发生了宕机，这种情况被称为部分写失效(partial page write)。
        + 如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚地认识到，重做日志中记录的是对页的物理操作，如偏移量800，写’aaaa’记录。如果这个页本身已经发生了损坏，再对其进行重做是没有意义的。
        + 流程
        + 在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是会通过memcpy函数将脏页先复制到内存中的doublewrite buffer，之后通过doublewrite buffer再分两次，每次1MB顺序地写入共享表空间的物理磁盘上，然后马上调用fsync函数，同步磁盘，避免缓冲写带来的问题。在这个过程中，因为doublewrite页是连续的，因此这个过程是顺序写的，开销并不是很大。在完成doublewrite页的写入后，再将doublewrite buffer中的页写入各个表空间文件中，此时的写入则是离散的
        + 命令

查看doublewrite运行情况

mysql＞SHOW GLOBAL STATUS LIKE&#39;innodb\_dblwr%&#39;\G;

doublewrite开启与关闭

mysql> show variables like &#39;innodb\_doublewrite&#39;;

* + - * + 注意事项

master-slaves

如果用户有多个从服务器（slave server），需要提供较快的性能（如在slaves erver上做的是RAID0），也许启用这个参数是一个办法。不过对于需要提供数据高可靠性的主服务器（master server），任何时候用户都应确保开启doublewrite功能

文件系统提供防范写失效

有的文件系统本身就提供了部分写失效的防范机制，如ZFS文件系统。在这种情况下，用户就不要启用doublewrite了

* + - * 自适应哈希索引
        + 定义
        + InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引（Adaptive Hash Index，AHI）
        + 特点

建立速度非常快

AHI是通过缓冲池的B+树页构造而来，因此建立的速度很快

不需要对整张表构建哈希索引

自动建立

InnoDB存储引擎会自动根据访问的频率和模式来自动地为某些热点页建立哈希索引

* + - * + 建立AHI条件

对这个页的连续访问模式必须是一样的

以该模式访问了100次

页通过该模式访问了N次，其中N=页中记录\*1/16

* + - * + 效果

读取速度

提升2倍

写入速度

提升2倍

辅助索引的连接性能

提升5倍

* + - * + 命令

AHI开启和关闭

mysql> show variables like &#39;innodb\_adaptive\_hash\_index&#39;;

查看AHI使用情况

mysql＞SHOW ENGINE INNODB STATUS\G;

* + - * + 注意事项
        + 哈希索引只能用来搜索等值的查询，如SELECT\*FROM table WHERE index\_col=‘xxx’。而对于其他查找类型，如范围查找，是不能使用哈希索引的
      * 异步IO
        + 分类

同步IO

即每进行一次IO操作，需要等待此次操作结束才能继续接下来的操作

异步IO

发出一个IO请求后立即再发出另一个IO请求，当全部IO请求发送完毕后，等待所有IO操作的完成.

Native AIO

内核级别提供的AIO支持

* + - * + 异步IO优势

缩短了IO时间

用户可以在发出一个IO请求后立即再发出另一个IO请求，当全部IO请求发送完毕后，等待所有IO操作的完成，缩短了时间

IO合并

可以进行IO Merge操作，也就是将多个IO合并为1个IO，这样可以提高IOPS的性能.

例如用户需要访问页的（space，page\_no）为：（8，6）、（8，7），（8，8）

每个页的大小为16KB，那么同步IO需要进行3次IO操作。而AIO会判断到这三个页是连续的（显然可以通过（space，page\_no）得知）。因此AIO底层会发送一个IO请求，从（8，6）开始，读取48KB的页。

* + - * + Native AIO

开启

通过innodb\_use\_native\_aio参数来控制

* + - * + 在InnoDB中作用

read ahead

脏页刷新

磁盘写入

* + - * 刷新邻接页
        + 定义
        + 当刷新一个脏页时，InnoDB存储引擎会检测该页所在区（extent）的所有页，如果是脏页，那么一起进行刷新。
        + 优势
        + 通过AIO可以将多个IO写入操作合并为一个IO操作，故该工作机制在传统机械磁盘下有着显著的优势
        + 开启和关闭

命令

通过innodb\_flush\_neighbors进行控制

注意事项

对于传统机械硬盘建议启用该特性，而对于固态硬盘有着超高IOPS性能的磁盘，则建议将该参数设置为0，即关闭此特性

* + - Innodb Checkpoint机制
      * 为什么需要Checkpoint机制
      * 为了协调CPU和硬盘之前的速度鸿沟,数据的操作首先是在缓冲池中进行的，然后再以一定的策略刷新的硬盘中。这就有两个问题:1.如果数据一有更新就刷新到磁盘，那数据库的效率太低了，不可取;2. 如果在还没有刷新到硬盘的过程中,数据库宕机了，数据就有可能丢失.综合上面两个原因，就需要Checkpoint机制
      * 定义
      * 白话说以一定的策略将检查点之前的数据全部刷入硬盘，而检查点之后的数据依然在缓冲池中。数据恢复直接读取检查点之前的数据，然后通过redo日志恢复当前检查点到宕机时刻的数据
      * 解决了哪些问题
        + 缩短了数据库的恢复时间
        + 当数据库发生宕机时，数据库不需要重做所有的日志，因为Checkpoint之前的页都已经刷新回磁盘。故数据库只需对Checkpoint后的重做日志进行恢复。这样就大大缩短了恢复的时间。
        + 缓冲池不够用时，将脏页刷新到磁盘
        + 当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Checkpoint，将脏页也就是页的新版本刷回磁盘。
        + 重做日志不可用时，刷新脏页
        + 重做日志出现不可用的情况是因为当前事务数据库系统对重做日志的设计都是循环使用的，并不是让其无限增大的，这从成本及管理上都是比较困难的。重做日志可以被重用的部分是指这些重做日志已经不再需要，即当数据库发生宕机时，数据库恢复操作不需要这部分的重做日志，因此这部分就可以被覆盖重用。若此时重做日志还需要使用，那么必须强制产生Checkpoint，将缓冲池中的页至少刷新到当前重做日志的位置
      * 种类
        + Sharp Checkpoint

参数控制

mysql> SHOW VARIABLES LIKE &#39;innodb\_fast\_shutdown&#39;;

作用

Sharp Checkpoint发生在数据库关闭时将所有的脏页都刷新回磁盘，这是默认的工作方式

* + - * + Fuzzy Checkpoint

作用

但是若数据库在运行时也使用Sharp Checkpoint，那么数据库的可用性就会受到很大的影响。故在InnoDB存储引擎内部使用Fuzzy Checkpoint进行页的刷新，即只刷新一部分脏页，而不是刷新所有的脏页回磁盘。

发生的情况

Master Thread Checkpoint

差不多以每秒或每十秒的速度从缓冲池的脏页列表中刷新一定比例的页回磁盘

FLUSH\_LRU\_LIST Checkpoint

InnoDB存储引擎需要保证LRU列表中需要有差不多100个空闲页可供使用。通过参数innodb\_lru\_scan\_depth控制LRU列表中可用页的数量

Async/Sync Flush Checkpoint

指的是重做日志文件不可用的情况，这时需要强制将一些页刷新回磁盘

Dirty Page too much Checkpoint

即脏页的数量太多，导致InnoDB存储引擎强制进行Checkpoint。其目的总的来说还是为了保证缓冲池中有足够可用的页。其可由参数innodb\_max\_dirty\_pages\_pct控制

* + 日志
    - 二进制日志
      * 定义
      * 以一定格式记录了对MySQL数据库执行更改的所有操作，但是不包括SELECT和SHOW这类操作，因为这类操作对数据本身并没有修改。然而，若操作本身并没有导致数据库发生变化，那么该操作可能也会写入二进制日志。
      * 作用
        + 恢复
        + 某些数据的恢复需要二进制日志，例如，在一个数据库全备文件恢复后，用户可以通过二进制日志进行point-in-time的恢复。
        + 复制
        + 其原理与恢复类似，通过复制和执行二进制日志使一台远程的MySQL数据库（一般称为slave或standby）与一台MySQL数据库（一般称为master或primary）进行实时同步。
        + 审计
        + 用户可以通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入的攻击
      * 配置
        + 路径
        + mysql> show variables like &#39;datadir&#39;;
        + +---------------+-----------------+
        + | Variable\_name | Value |
        + +---------------+-----------------+
        + | datadir | /var/lib/mysql/ |
        + +---------------+-----------------+
        + 配置流程
        + //二进制文件默认没有开启:
        + mysql> show variables like &#39;%bin%&#39;;
        + | log\_bin | OFF |
        + | log\_bin\_basename | |
        + | log\_bin\_index | |
        + //配置文件加入配置项
        + [mysqld]
        + log\_bin=/var/log/mysql/binlog.log
        + server-id=1 //这里需要注意
        + //保存 关闭配置文件 重启mysql
        + service mysql restart
        + //重启登录mysql查看binlog信息
        + mysql> show variables like &#39;%bin%&#39;;
        + | log\_bin | ON |
        + | log\_bin\_basename | /var/log/mysql/binlog |
        + | log\_bin\_index | /var/log/mysql/binlog.index |
        + ————————————————
      * 查看
        + 相关参数
        + mysql> show variables like &#39;%bin%&#39;;
        + 查看二进制日志状态
        + mysql> show master status;
        + mysql> SHOW BINLOG EVENTS IN &#39;binlog.000002&#39;;
        + mysqlbinglog查看
        + root@ubuntu:/var/log/mysql# mysqlbinlog -vv --start-position=1094 binlog.000002
      * 弊端
        + 影响性能
        + 虽影响性能,但十分有限,开启二进制日志会使性能下降1%
        + 占用磁盘空间
        + binlog\_format参数设置为ROW，会加大二进制文件的大小。每次执行相关的语句，会占用用磁盘空间
    - 慢查询日志
      * 定义
      * 将时间大于一定阈值的sql语句记录在某一个文件中，这个文件就是慢查询日志
      * 配置
        + 开启
        + //开启慢查询日志
        + mysql> set global slow\_query\_log=ON;
        + mysql> show variables like &#39;slow\_query%&#39;;
        + 设置sql语句超时时间
        + //设置sql语句超时时间
        + mysql> show variables like &#39;long\_query\_time&#39;;
        + mysql> set session long\_query\_time=1;
        + mysql> set global long\_query\_time=1;
        + mysql> show variables like &#39;long\_query\_time&#39;;
        + mysql> select @@global.long\_query\_time;
        + mysql> select @@session.long\_query\_time;
        + 开启未使用索引查询的sql语句
        + //开启未使用索引查询的sql语句
        + mysql> set global log\_queries\_not\_using\_indexes=ON;
        + mysql> show variables like &#39;log\_queries\_not\_using\_indexes&#39;;
        + //每分钟允许记录到slow log的且未使用索引的SQL语句次数:0 无限制
        + mysql> show variables like &#39;log\_throttle\_queries\_not\_using\_indexes&#39;;
      * 慢查询日志表:slow\_log
        + 查看
        + //mysql.slow\_log 表信息
        + mysql> SHOW CREATE TABLE mysql.slow\_log;
        + 将慢查询输出格式由文件转化为表
        + mysql> SHOW VARIABLES LIKE&#39;log\_output&#39;;
        + mysql> SET GLOBAL log\_output=&#39;TABLE&#39;;
        + mysql> SHOW VARIABLES LIKE&#39;log\_output&#39;;
        + 提高在大数据下查询效率
        + 将mysql.slow\_log表存储引擎由CSV修改为MyISAM,并在start\_time列上添加索引以进一步提高在大数据下查询的效率。不能忽视的是，将slow\_log表的存储引擎更改为MyISAM后，还是会对数据库造成额外的开销。不过好在很多关于慢查询的参数都是动态的，用户可以方便地在线进行设置或修改。
    - 重做日志文件
      * 重做日志缓冲
      * InnoDB产生的日志不可能实时刷新到磁盘中，这样效率太差，所以都时先放入内存的缓冲区中，然后以一定的规则刷新到磁盘中。这个缓冲buffer就是重做日志缓冲
      * 定义
      * 在InnoDB数据目录下存有ib\_logfile0和ib\_logfile1,这2个文件就是重做日志文件,记录了InnoDB存储引擎的事务日志，记录了每个页更改的物理情况
      * 作用
      * 当实例或者介质失效时，InnoDB存储引擎会根据重做日志恢复到实例或者介质失效前的那一刻状态
      * 文件属性
      * mysql> SHOW VARIABLES LIKE&#39;innodb%log%&#39;;
        + innodb\_log\_file\_size
        + 重做日志文件大小
        + innodb\_log\_files\_in\_group
        + 指定了日志文件组中，重做日志文件的数量,默认为2
        + innodb\_mirrored\_log\_groups
        + 指定了日志镜像文件组的数量，默认为1,表示只有一个日志文件组，没有镜像。若磁盘本身已经做了高可用的方案，如磁盘阵列，那么可以不开启重做日志镜像的功能
        + innodb\_log\_group\_home\_dir
        + 指定了日志文件组所在的目录
      * 文件大小
        + 太大
        + 在恢复时可能需要很长的时间
        + 太小
        + 可能导致一个事务的日志需要多次切换重做日志文件
      * 写入磁盘
        + 写入流程
        + 先写入一个重做日志缓冲（redo log buffer）中，然后按照一定的条件顺序地写入日志文件。从重做日志缓冲往磁盘写入时，是按512个字节，也就是一个扇区的大小进行写入。因为扇区是写入的最小单位，因此可以保证写入必定是成功的。因此在重做日志的写入过程中不需要有doublewrite

先写入redo log buffer

按512个字节依次写入磁盘

* + - * + 写入磁盘条件

主线程

主线程(master thread)中每秒会将重做日志缓冲写入磁盘的重做日志文件中，不论事务是否已经提交

innodb\_flush\_log\_at\_trx\_commit

mysql> show variables like &#39;innodb\_flush\_log\_at\_trx\_commit&#39;;

0

代表当提交事务时，并不将事务的重做日志写入磁盘上的日志文件，而是等待主线程每秒的刷新。

1

表示在执行commit时将重做日志缓冲同步写到磁盘，即伴有fsync的调用。为了保证事务的ACID中的持久性，必须将innodb\_flush\_log\_at\_trx\_commit设置为1

2

表示将重做日志异步写到磁盘，即写到文件系统的缓存中。因此不能完全保证在执行commit时肯定会写入重做日志文件，只是有这个动作发生。

* + - * 文件组
      * log group为重做日志组，其中有多个重做日志文件。
      * 恢复机制
        + 原理
        + InnoDB存储引擎在启动时不管上次数据库运行时是否正常关闭，都会尝试进行恢复操作。因为重做日志记录的是物理日志，因此恢复的速度比逻辑日志，如二进制日志，要快很多
        + 恢复流程
        + 由于checkpoint表示已经刷新到磁盘页上的LSN，因此在恢复过程中仅需恢复checkpoint开始的日志部分。对于图7-12中的例子，当数据库在checkpoint的LSN为10 000时发生宕机，恢复操作仅恢复LSN 10 000～13 000范围内的日志。
      * 记录的内容
      * InnoDB存储引擎的重做日志是物理日志，因此其恢复速度较之二进制日志恢复快得多。例如对于INSERT操作，其记录的是每个页上的变化。记录的是页的物理修改操作
      * 重做日志(redo log) VS 二进制日志(binlog)
        + 定义

二进制日志

其用来进行POINT-IN-TIME（PIT）的恢复及主从复制（Replication）环境的建立

重做日志

在InnoDB数据目录下存有ib\_logfile0和ib\_logfile1,这2个文件就是重做日志文件,记录了InnoDB存储引擎的事务日志，记录了每个页更改的物理情况

* + - * + 产生方式
        + 重做日志是在InnoDB存储引擎层产生，而二进制日志是在MySQL数据库的上层产生的，并且二进制日志不仅仅针对于InnoDB存储引擎，MySQL数据库中的任何存储引擎对于数据库的更改都会产生二进制日志。

二进制日志

MySQL数据库的上层产生

重做日志

InnoDB存储引擎层产生

* + - * + 记录内容
        + 两种日志记录的内容形式不同。MySQL数据库上层的二进制日志是一种逻辑日志，其记录的是对应的SQL语句。而InnoDB存储引擎层面的重做日志是物理格式日志，其记录的是对于每个页的修改

二进制日志

逻辑日志

重做日志

物理日志

* + - * + 写入磁盘的时间点
        + 二进制日志只在事务提交完成后进行一次写入。而InnoDB存储引擎的重做日志在事务进行中不断地被写入，这表现为日志并不是随事务提交的顺序进行写入的。

二进制日志

事务提交一次性写入

重做日志

随着事务进行,不断写入

* + - undo log
      * 定义
      * 记录了事务的运行日志,用来进行事务回滚的所需要的文件
      * 作用
        + 对事务进行回滚
        + MVCC控制
        + 当用户读取一行记录时，若该记录已经被其他事务占用，当前事务可以通过undo log读取之前的行版本信息，以此实现一致性非锁定读取。
      * 存放位置
        + 共享表空间内的undo log段
      * 注意事项
        + undo log是逻辑日志
        + undo是逻辑日志，因此只是将数据库逻辑地恢复到原来的样子。所有修改都被逻辑地取消了，但是数据结构和页本身在回滚之后可能大不相同。
        + undo log会产生redo log
        + undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护
      * 存储管理
        + undo 设置
        + mysql> show variables like &#39;innodb\_undo%&#39;;
        + mysql> show variables like &#39;datadir%&#39;;
        + 段管理
        + InnoDB存储引擎有rollback segment，每个回滚段种记录了1024个undo log segment，而在每个undo log segment段中进行undo页的申请。
        + 申请与写入

undo页申请

undo log segment段中进行undo页的申请.分配和申请需要产生redo log

事务提交时

1. 将undo log放入列表中，以供之后的purge操作

2. 判断undo log所在的页是否可以重用，若可以分配给下个事务使用

事务提交后

事务提交后并不能马上删除undo log及undo log所在的页。这是因为可能还有其他事务需要通过undo log来得到行记录之前的版本。故事务提交时将undo log放入一个链表中，是否可以最终删除undo log及undo log所在页由purge线程来判断。

* + - * + undo log重用

作用:减少占用磁盘空间

若为每一个事务分配一个单独的undo页会非常浪费存储空间，特别是对于OLTP的应用类型。因为在事务提交时，可能并不能马上释放页。假设某应用的删除和更新操作的TPS（transaction per second）为1000，为每个事务分配一个undo页，那么一分钟就需要1000\*60个页，大约需要的存储空间为1GB。若每秒的purge页的数量为20，这样的设计对磁盘空间有着相当高的要求。因此，在InnoDB存储引擎的设计中对undo页可以进行重用。

重用算法

当事务提交时，首先将undo log放入链表中，然后判断undo页的使用空间是否小于3/4，若是则表示该undo页可以被重用，之后新的undo log记录在当前undo log的后面

* + - * + undo log回收

相关设置

//每次purge操作需要清理的undo page数量

mysql> show variables like &#39;innodb\_purge\_batch\_size&#39;;

//用来控制history list的长度，若长度大于该参数时，其会“延缓”DML的操作。该参数默认值为0，表示不对history list做任何限制。

mysql> show variables like &#39;innodb\_max\_purge\_lag&#39;;

//当innodb\_max\_purge\_lag 大于0时，就会延缓DML的操作，其延缓的算法为：delay=（（length(history\_list)-innodb\_max\_purge\_lag）\*10)-5

//来控制delay的最大毫秒数。也就是当上述计算得到的delay值大于该参数时，将delay设置为innodb\_max\_purge\_lag\_delay，避免由于purge操作缓慢导致其他SQL线程出现无限制的等待。

mysql> show variables like &#39;innodb\_max\_purge\_lag\_delay&#39;;

算法概述

InnoDB存储引擎这种通过purge线程先从history list中找undo page，然后再从undo page中找undo log

优点

避免大量的随机读取操作，从而提高purge的效率

* + - * undo log分类
        + insert undo log

定义

是指在insert操作中产生的undo log

回收

因为insert操作的记录，只对事务本身可见，对其他事务不可见（这是事务隔离性的要求），故该undo log可以在事务提交后直接删除。不需要进行purge操作

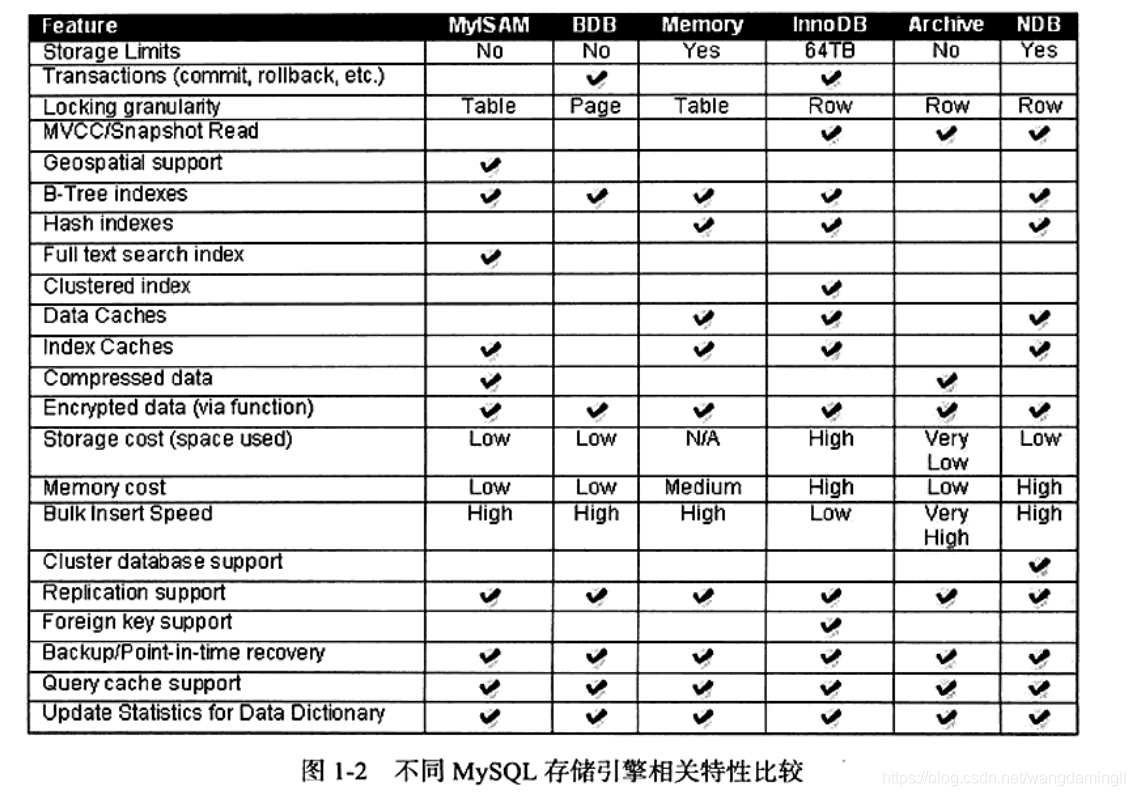
* + - * + update undo log

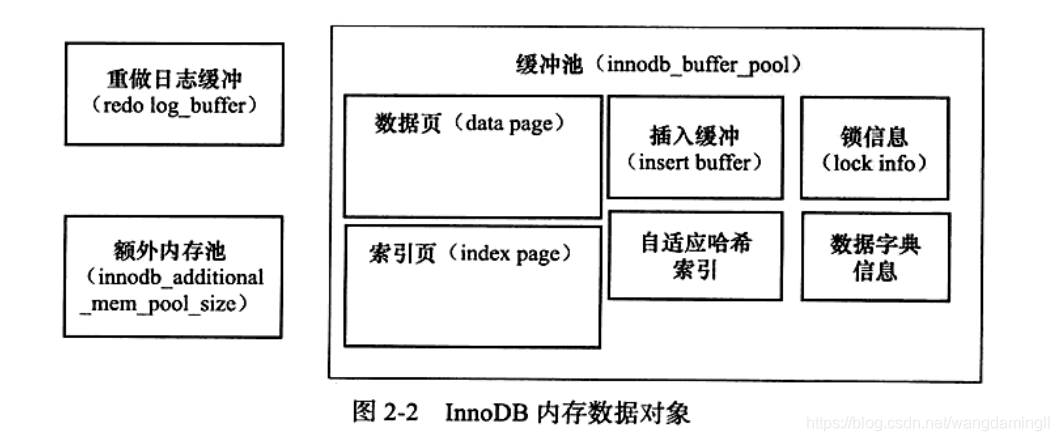
定义

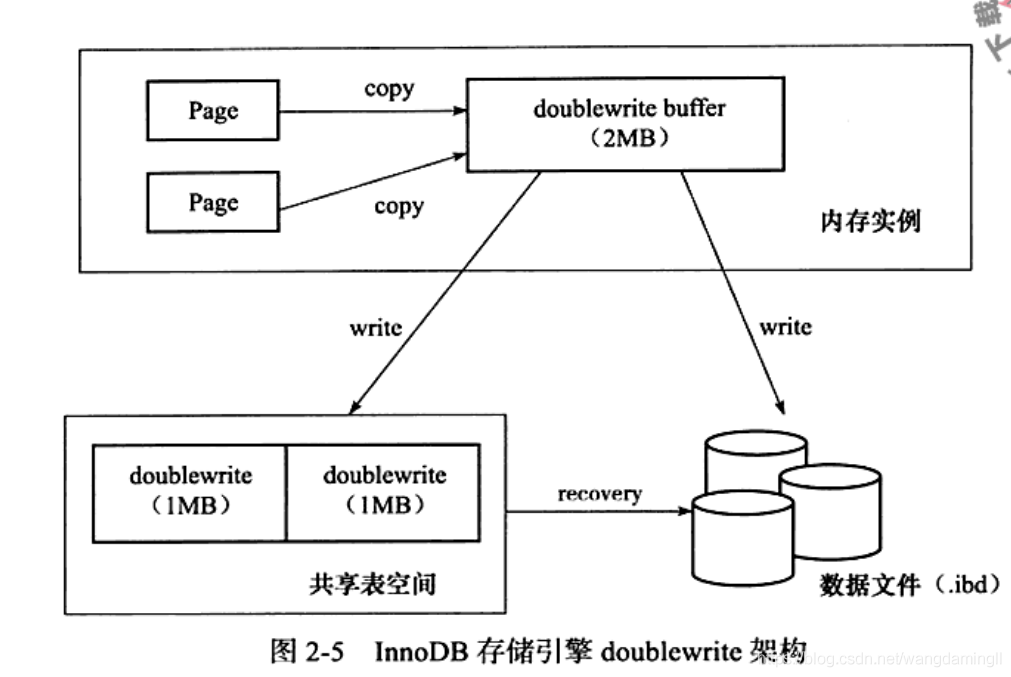
记录的是对delete和update操作产生的undo log

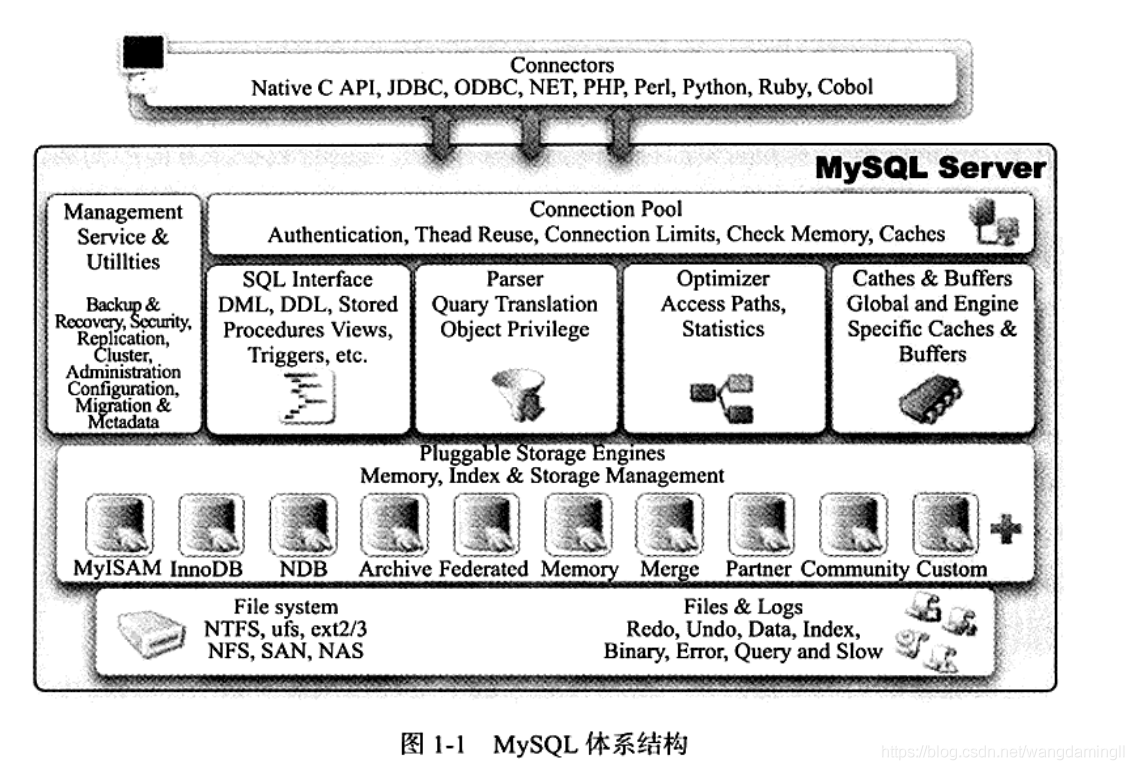
回收

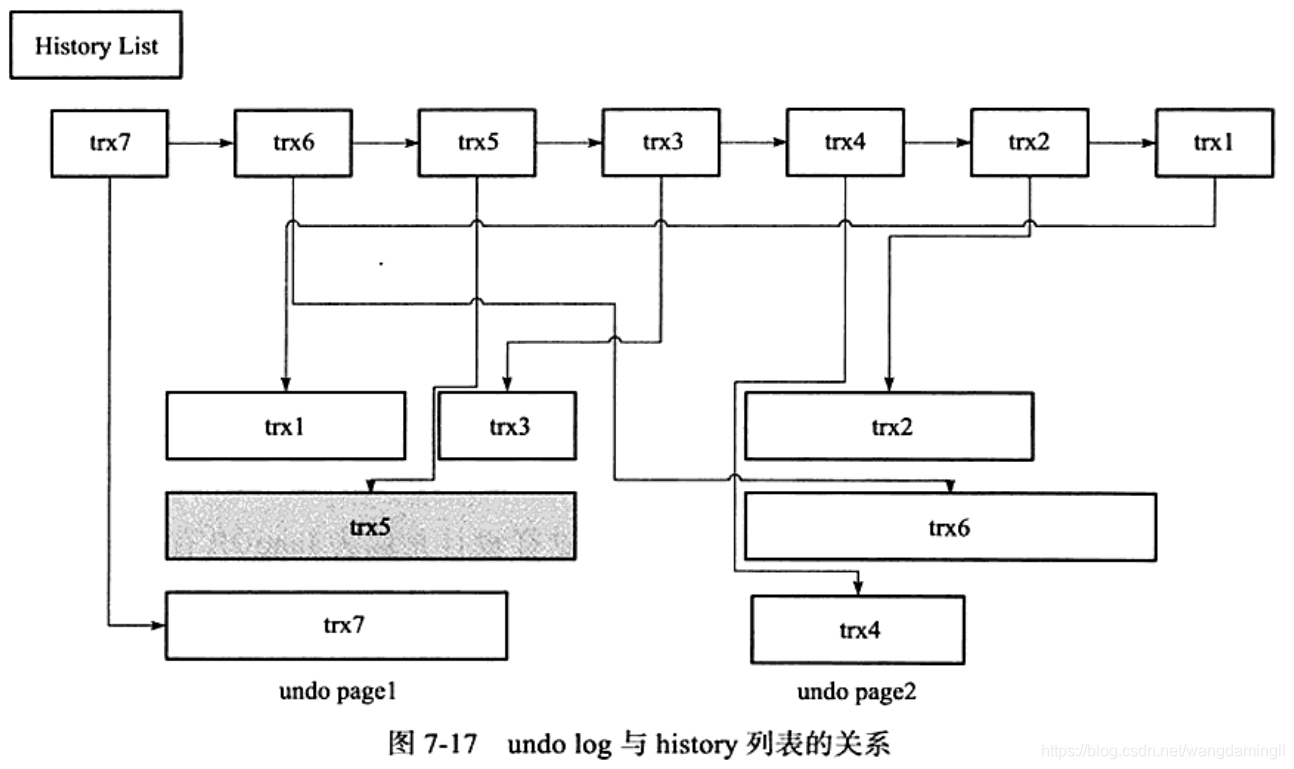
该undo log可能需要提供MVCC机制，因此不能在事务提交时就进行删除。提交时放入undo log链表，等待purge线程进行最后的删除。

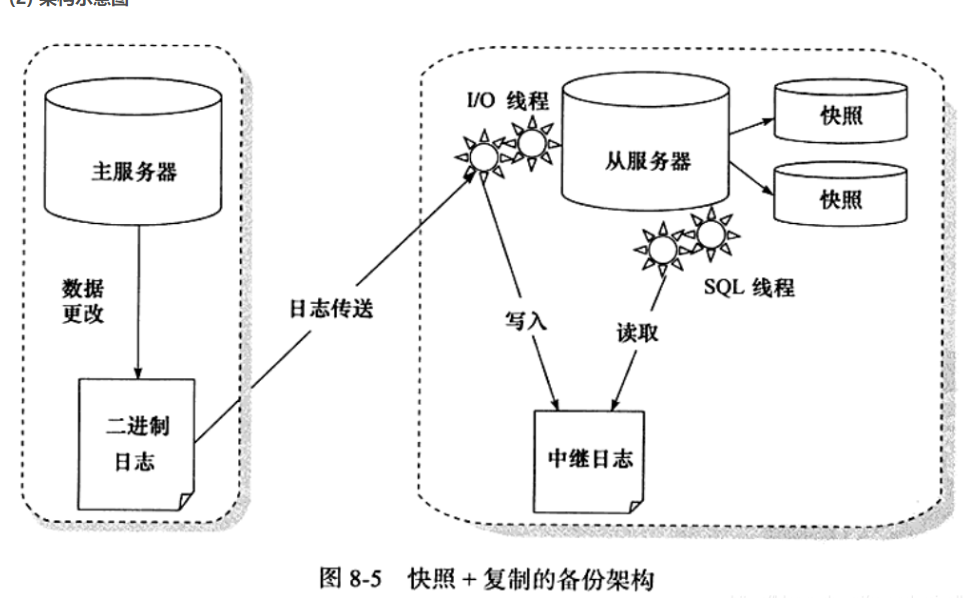








  
复制 + 快照备份架构

  
恢复流程

