MySQL 事务、日志、锁、索引学习总结

MySQL 架构

MySQL 可分为 Server 和存储引擎两部分，如图 1 所示。
Server 层：包括客户端连接器、查询缓存、解析/预处理器、优化器、执行器等，以及 MySQL 内置函数和所有跨引擎的功能都在这一层实现，比如存储过程、触发器、视图等。
存储引擎层：负责数据的存储和读取，为插件式架构，支持 innoDB、MyISAM、Memory 等多个存储引擎，InnoDB 为默认存储引擎。

图 1 MySQL 逻辑架构

表存储

表包含两部分，表结构定义和数据。在 MySQL 中，表由存储引擎负责存储，以 InnoDB 为例：

1. 表结构定义：在 MySQL8.0 之前，只能存在于.frm 后缀文件中(MySQL Server 层和 InnoDB 中均存在)；之后允许将其放在系统数据表中。
2. 表数据：包括数据段(主键索引)和索引段(二级索引)，由 **innodb_file_per_table** 参数控制存储位置，自 5.6.6 开始，默认配置为 ON。
    1. OFF 表示，存放在系统共享表空间
    2. ON 表示，单独存放在一个.ibd 文件中

使用单独文件存储表数据，在删除表时直接删除文件可以回收空间。
而在共享表空间中存储，即使将表删除，空间也是不会回收的。因为 InnoDB 中数据以 B+树结构组织，删除其中一部分记录，只是将其相应的位置标记为删除可复用(由于索引排序，记录只能被相应的数据所复用)；而当数据页上所有记录都被删除时，意味着数据页可以复用到任意位置。

在删除数据之后而又没用复用时，就会造成数据空洞；同样的，在新增时也可能导致数据空洞，当一个数据页 A 已经写满了，但由于索引的有序性，需要在 A 中间再插入一条数据，这时就需要进行页分裂再申请一个数据页来保存数据(当前数据以及之后的索引分裂到新的数据页)。

在大量增删操作之后的表可能存在数据空洞，即很多位置无法复用。通过重建表可以实现空间收缩：

1. recreate table: alter table t engine = InnoDB，可以实现在线重建表，短暂持有 MDL 写锁，之后持有 MDL 读锁。使用一个 rowlog 存储重建表期间表数据的修改记录，不会阻塞其他事务的增删改。

2. analyze table t 对表的索引信息做重新统计，没有修改数据，加 MDL 读锁。
3. optimize table t 等于 recreate+analyze。

日志

MySQL 中记录日志的方式为 WAL(Write-Ahead Logging），先预写日志再更新数据，对于非内存数据库来说，可以减少磁盘 IO 提高性能。
MySQL 日志：

1. binlog：在对数据进行增删改之后，都将会记录一条 binlog,可用于数据归档和备份，存在两种格式的 binlog_format：

    1. statement 记录的是 SQL 语句，最后会有 COMMIT。
    2. row 记录的实际操作的数据记录，最后会有一个 XID event。

sync_binlog 设置为 1 时，表示每次事务操作的 binlog 都持久化到磁盘中，在 MySQL 异常重启后可保证 binlog 不丢失。

InnoDB 日志：

1. redolog：在对数据进行增删改之后，都将会记录一条 redolog。其为物理日志，记录的是在某个数据页上做了什么修改，**可用于崩溃后恢复事务数据和减少更新数据时的磁盘 IO 访问**。innodb_flush_log_at_trx_commit 这个参数设置成 1 的时候，表示每次事务的 redo log 都直接持久化到磁盘。
2. undolog：在事务中对数据每进行一次修改便会记录一次 undolog，用于将最新数据恢复到之前事务版本。在长事务中可能占用大量存储空间。在系统判定 undo-log 无用时，会将其删除，即在没有比回滚日志更早的 Read View 时。

binlog 和 redolog 存在一个共同的数据字段 XID，通过这个字段可以将 redolog 和 binlog 关联起来，可用于事务恢复。

索引

在 InnoDB 中，表数据都是根据主键顺序以索引的形式存放的，这种存储方式的表称为索引组织表。索引的底层数据结构为 B+树，所以每一个索引在 InnoDB 中都对应一颗 B+树，InnoDB 中存在有两种类型的索引：

1. 聚簇索引(主键)
聚簇索引的叶子结点存的是整行数据。
2. 二级索引
二级索引分为唯一和普通索引，叶子结点中存的是主键的值，如果需要获取整行数据，需要使用主键值再去聚簇索引中回表查询。

索引维护：由于底层数据结构为 B+树，所以维护索引就是在维护 B+树；而 B+树是有序的，插入更新数据时可能导致数据移动而引入额外磁盘 IO。而在索引

字段重复时，又会页分裂出新的数据页来保存重复 Key。

B+ 树的高和阶：阶由页大小(默认 16K)和索引大小而决定，而高又由阶和行数决定。

InnoDB 事务

数据库事务是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成，具备四个基本属性，原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Duarbility)。

如何启动/回滚事务：

1. 手动使用 BEGIN, ROLLBACK, COMMIT 来实现；BEGIN 开始一个事务，ROLLBACK 事务回滚，COMMIT 事务提交
2. 直接用 SET AUTOCOMMIT = 0/1 来改变 MySQL 的自动提交模式:

    1. 若参数 autocommit=0(禁止自动提交)，事务则在用户本次对数据进行操作时自动开启，在用户执行 commit 命令时提交，用户本次对数据库开始进行操作到用户执行 commit 命令之间的一系列操作为一个完整的事务周期。若不执行 commit 命令，系统则默认事务回滚。总而言之，当前情况下事务的状态是自动开启手动提交。
    2. 若参数 autocommit=1（系统默认值，开启自动提交），事务的开启与提交又分为两种状态：

        1. 手动开启手动提交：当用户执行 start transaction 命令时（事务初始化），一个事务开启，当执行 commit 命令时事务提交，若不执行 commit 命令，系统则默认事务回滚。
        2. 自动开启自动提交：如果用户在当前情况下未执行 start transaction 命令而对数据库进行了操作，系统则默认用户对数据库的每一个操作为一个孤立的事务，也就是说用户每进行一次操作系都会即时提交或者即时回滚。

事务提交:

InnoDB 中事务分为两阶段提交：

1. 第一阶段是在更新完数据后，记录 redo-log，这时 redolog 状态为 prepare
2. 第二阶段是在记完 redo-log 之后，记录 bin-log，将 redolog 状态置为 commit

两阶段提交常用于分布式系统中，InnoDB 中使用两阶段提交可以保证在事务恢复时，其 binlog 是正确的；如果只记录 redolog，在恢复事务之后就会导致数据与 binlog 不一致。

事务恢复

得益于两阶段提交，事务在恢复之后可以保证数据与 binlog 的一致，事务恢复时的判断规则为：

1. 如果 redo log 里面的事务是完整的，也就是有 prepare、commit 标识，则直接提交；
2. 如果 redo log 里面的事务只有完整的 prepare，则判断对应的事务 binlog 是否是存在并且完整的，如果是，则提交事务； 否则，回滚事务。

实现事务隔离

在 InndDB 中，通过 MVCC(一致性视图)实现事务的隔离性，在 InnoDB 中，一行数据物理上只保存最新值，但通过 undolog 可以回滚到之前事务版本，所以数据可能存在多个版本(事务 Id 用作区分)。MVCC 便是使用事务 Id、数据、undolog 来实现一致性视图(read-view) ，但在生成一致性读时，可重复读(Repeatable read)和读提交(read committed)的 read view 生成策略是不一致的：

1. 在可重复读级别下，整个事务存在期间都使用同一个视图，只会获取小于等于当前事务 Id 版本数据，如果数据被更新了，就通过 undolog 计算得到相应版本的数据，**解决了不可重复读问题**。
2. 在读提交级别下，视图在每条 SQL 执行期间创建，只获取已提交的最新事务版本数据，所以每条 SQL 看到的数据可能都是不一致的，**存在不可重复读问题**。

事务问题

1. 在可重复读级别下，每条 SQL 使用到的锁需要等到事务提交或回滚之后才释放，存在长事务时，可能会占用的更多的资源，如锁、undolog 等，所以应避免长事务并且将资源占用较多的 SQL 放在事务后程进行。
2. 在读提交级别下，每条 SQL 使用到的锁在 SQL 执行完成后便会释放，在多事务并行时，如果 binlog_format=statement 时可能造成数据和 binlog 的不一致，所以应将其设置为 row。

InnoDB 行锁、间隙锁、临键锁

行锁(record lock)：

InnoDB 事务中，一条更新语句执行时，必须要获得其行写锁，而行锁分为读锁和写锁，其中读锁之间兼容，读写锁、写锁之间互斥，如 select id from table1 lock in share mode 当前读加读锁；select if from table1 for update 当前读加写锁。在 RR 级别下，行锁在需要的时候才加上，但是得等到事务结束时才释放，这叫做两阶段锁协议，两阶段加锁协议主要为了保证事务的隔离性(解决不可重复读)和一致性(数据状态一致)。

间隙锁(Gap Lock)：

通过两阶段锁协议可以解决不可重复读和数据一致性问题，但幻读(两次当前读时，行数不一致)仍然存在，即便将所有行都加上行锁也无法解决幻读问题。间隙锁(Gap Lock)的引入就是为解决幻读问题，间隙锁锁定的记录之间间隙，是一个区间范围；在被间隙锁锁定的区间范围内，不能插入新的数据。
如索引中存在三个聚簇节点 Id[1,3,6]，事务 A 执行 update table t1 set name = 'ss' where t1.id = 3，若没有间隙锁，事务可以同时执行 insert into table(id, name) values(4, 's4')；但由于间隙锁的存在(1,6)，事务 B 需要等待事务 A 释放间隙锁之后才能新增成功。在当前读时，索引扫描到的记录都会加上间隙锁，区间为前开后开。

临键锁(next-key lock)：

临键锁是加锁的基本单位，由行锁+间隙锁组成，区间范围为前开后闭。在使用 select * from table for update 时，会将表中所有记录行锁和间隙都锁住，间隙锁区间为(−∞，+∞]。

加锁有两个基本原则，**一是加锁基本单位为临键锁；二是索引查找过程中访问到的对象才会加锁**。针对等值查询，有两个专门优化的点，**一是在唯一索引上加锁时，临键锁退化成行锁(唯一约束已经确保不能新增相同数据)；二是在普通索引上时，临键锁退化成间隙锁(间隙已经确保无法新增相同数据)**。

InnoDB Buffer

InnoDB 使用缓冲池(buffer pool)管理内存，在缓冲池中存在一个 change buffer，用来对在数据进行增、删、改时进行优化，可以减少随机 IO 读取。
innodb_change_buffer_max_size=50，表示 change buffer 最多占用 buffer bool 的 50%
InnoDB 中查询记录是一条一条的，但是读取时是以数据页为单位的，读取一条记录时会将记录所在的数据页整个读取到缓冲池中。

Change Buffer 的应用

数据的更新/插入/删除都包括对聚簇索引和普通索引的修改，这个过程中分为两种情况：

1. 如果对应数据页(聚簇和二级)已经存在于内存：直接更新内存中的数据页，记录 redo-log、binlog；
2. 如果数据页不在内存之中，对于唯一索引(包括聚簇)，需要将数据加载到内存中进行唯一性约束校验，校验通过再在内存中更新数据、记录 redolog；对于非唯一索引，直接将数据更改日志存储在 change-buffer 中(不写磁盘)，记录 redolog、binlog。

对于一个唯一、非唯一索引皆存的表来说，对于数据的更新可能是不同步的，主键、唯一索引数据已经在内存中更改、而普通索引则是将更改先记录在 change buffer 中，写入 change buffer 时也会记录 redolog、binlog，确保崩溃后恢复。change buffer 不仅在内存中，也会被写入到系统表空间中，change

buffer 写入磁盘的操作叫做 purge。在特定时刻，change buffer 中的操作会 merge 到原数据页中：一是访问这个数据页时；二是后台线程定期维护；三是在系统正常关闭的过程中。