InnoDB存储引擎

UPDATE语句的执行过程

2023-09 @灿太

目录

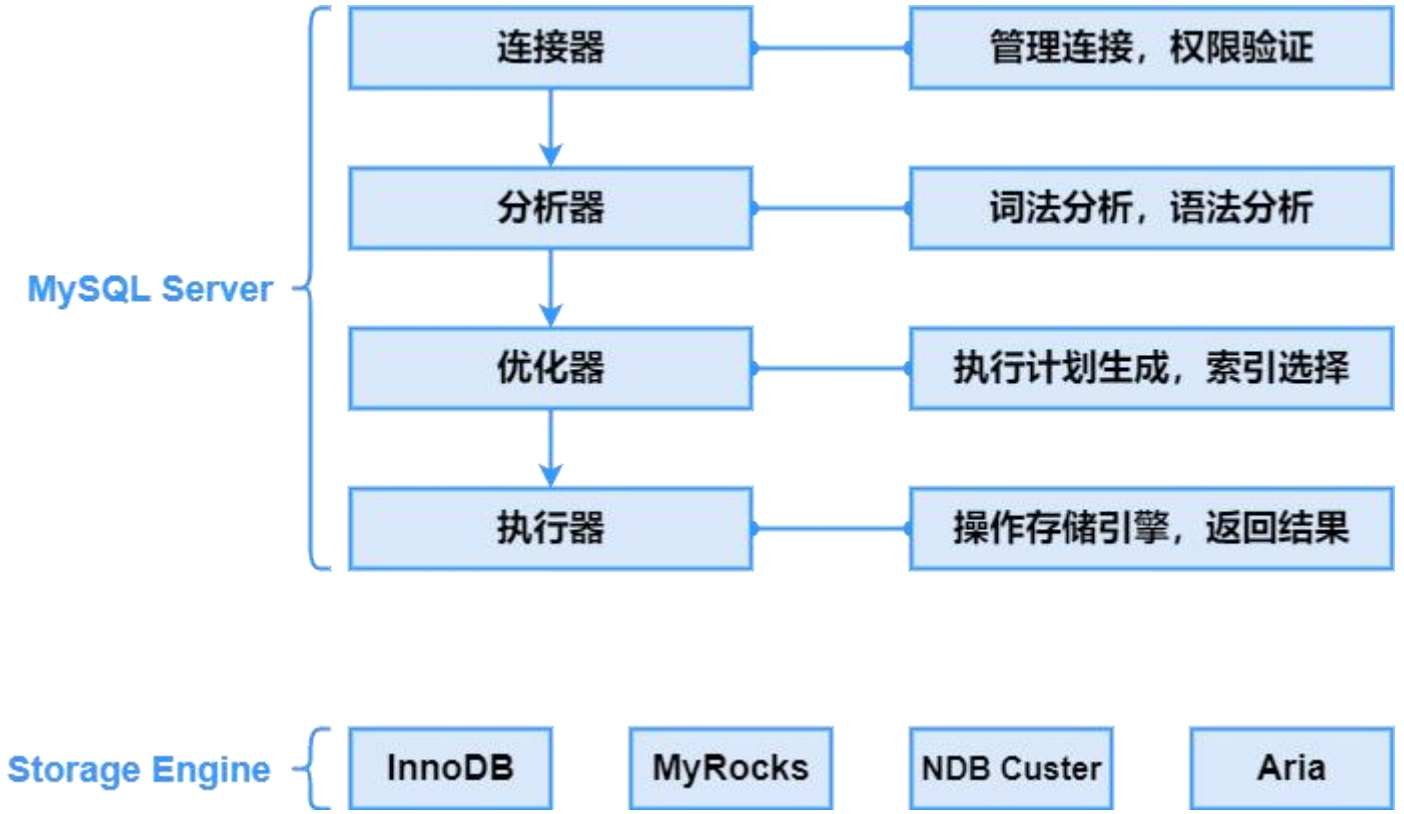
1. 开启事务

```
start transaction;
update t set status = 1 where name = 'cantai';
commit;
```

- 默认情况下 autocommit=1，如果我们没有显示开启事务，而是直接执行 update，那么 MySQL 就会隐式开启一个事务，并在这条 update 执行结束后自动提交。
- 如果我们 set autocommit=0，或者执行了 begin / start transaction 显示开启事务，那么update 执行完成后不会自动提交，需要手动发起 commit / rollback。

2. 解析SQL、生成查询计划

SQL执行流程



2. 解析SQL、生成查询计划

生成查询计划

表的访问顺序	确定查询涉及的表的访问顺序，最小化中间结果集的大小。
索引选择	确定选择一级、二级还是联合索引进行查询，以及哪些索引可以加速。
连接方法	如果查询涉及多表，需要确定如何连接：嵌套循环连接、哈希连接或者合并连接。
过滤条件和排序要求	考虑过滤条件和排序要求，如何优化地应用这些条件。
访问路径	确定如何访问表的数据，访问顺序，索引查询或者全表扫描。
使用临时表	确定是否需要创建和使用临时表来存储中间结果。

3. 查询数据

InnoDB 为了加速与磁盘的交互，设计了 Buffer Pool 缓冲池，存储内容如下：

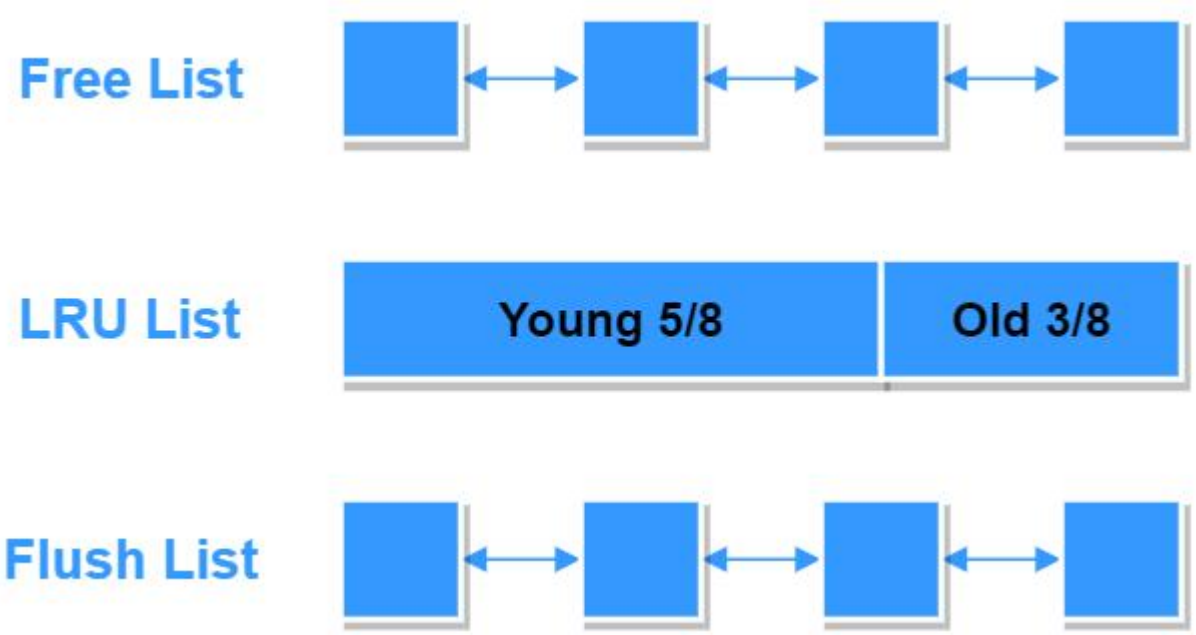
Buffer Pool 为每一个缓存页都创建了一个描述块，记录了缓存页的 namespace ID、page number、缓存页地址，描述块与缓存页一一对应。

Buffer Pool 同时使用一个哈希表 Page Table 加速对缓存页地址的检索，使用 namespace ID、page number 作为 key，控制块的地址作为 value。

3. 查询数据

Buffer Pool 中有三条链表：

1. Free List：空闲链，负责管理未被使用的缓冲池空间。
2. LRU List：最近最少使用链，负责在缓冲池满时淘汰缓冲页。
3. Flush List：脏链，主要负责管理要被刷新到磁盘的页。

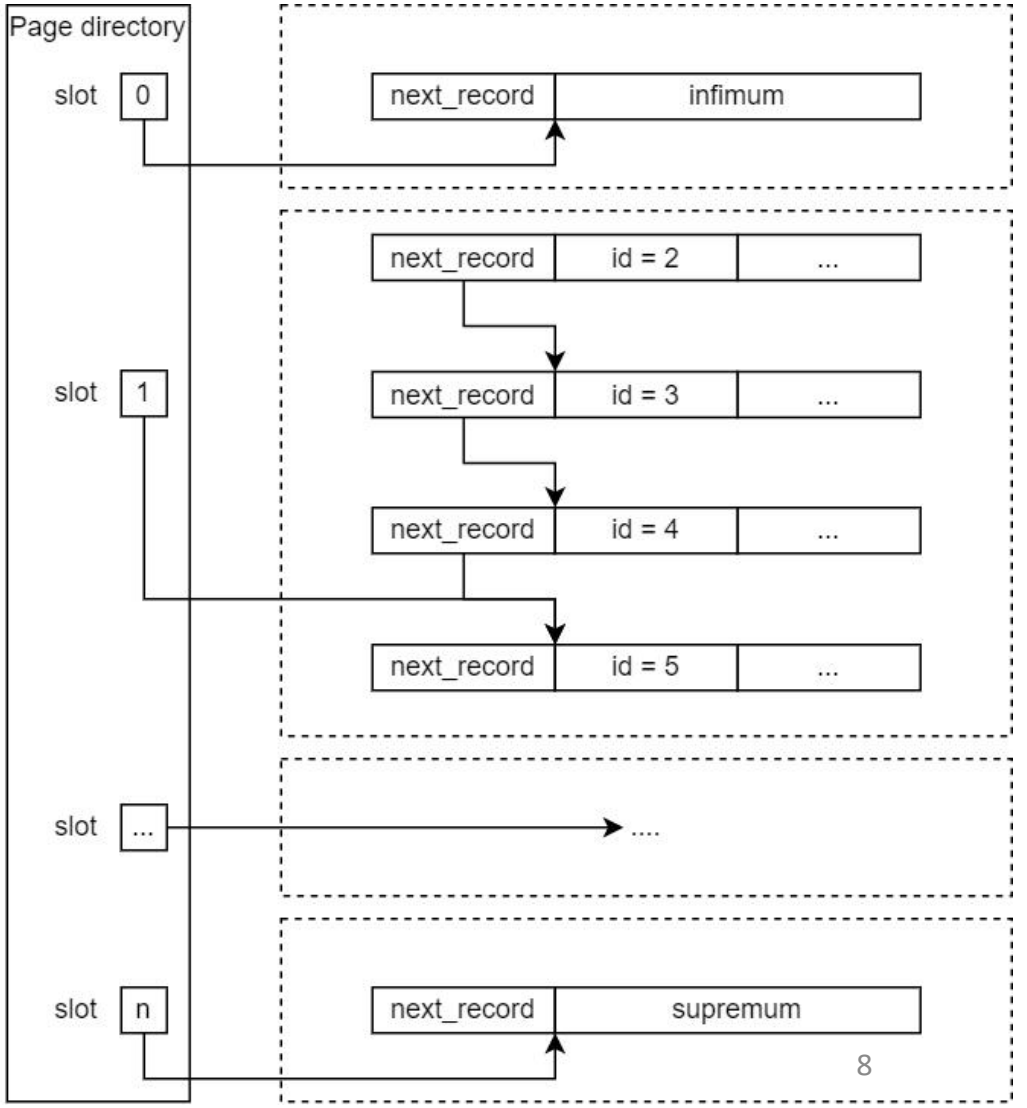


3. 查询数据

数据页结构

用户记录 User Records 根据索引排序，会被分组，每组的最大记录的地址偏移量提取出来，从 File Trailer 往前写，每个地址占用两个字节，称作槽，形成 Page Directory。

查询记录时先从 Page Directory 二分查找，定位记录所在的组，再遍历组，提高效率。



4. 校验锁和加锁

数据库事务隔离发展史

- 1992 年，ANSI 定义了异象标准，并根据所排除的异象，定义了，Read Uncommitted、Read Committed、Repeatable Read、Serializable四个隔离级别；
- *http://www.adp-gmbh.ch/ora/misc/isolation_level.html*

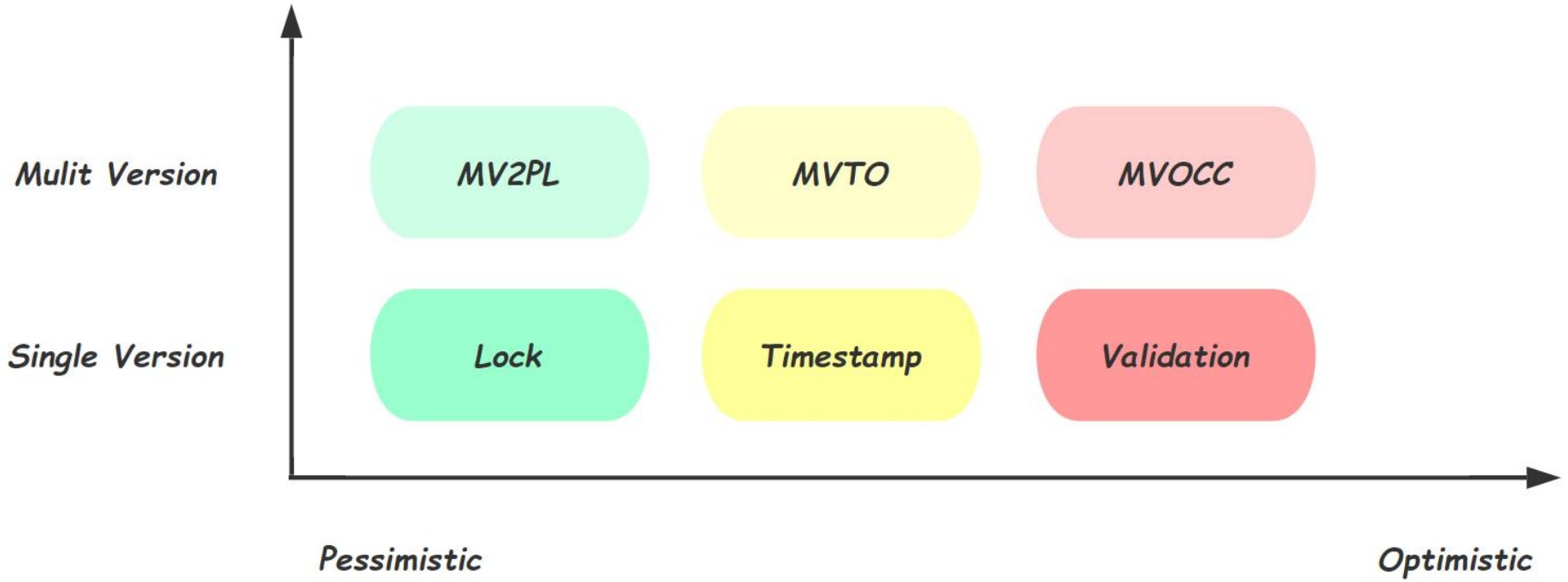
- 1995 年，微软的研究员选择用更严格的基于Lock的定义扩大了每个级别限制的范围；
- *https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf*

- 1999年，A Generalized Theory 认为基于Lock的定义过多的扩大了限制的范围，并给出了基于序列化图的定义方式，将每个级别限制的范围最小化。
- *https://pmg.csail.mit.edu/papers/adya-phd.pdf*

4. 校验锁和加锁

事务的并发控制方式

1. Two-phase Locking (MV2PL) — InnoDB
2. Timestamp Ordering (MVTO)
3. Optimistic Concurrency Control (MVOCC)



4. 校验锁和加锁

AREIS/KVL 论文提出一套完整的、高并发的实现算法，引导了B+Tree加锁领域几十年的研究和工业实现。

- Lock ：隔离多个事务，实现ACID特性。锁定数据库的逻辑内容，支持复杂的调度策略。
- Latch：保护内存中数据结构，实现线程安全，轻量级，通过规定的顺序申请以避免死锁。

	Lock	Latch
隔离级别	用户事务	线程
保护对象	数据库内容	内存数据结构
持续时间	整个事务周期	临界区代码前后
死锁检测	监测并解决	避免死锁出现

4. 校验锁和加锁-Latch

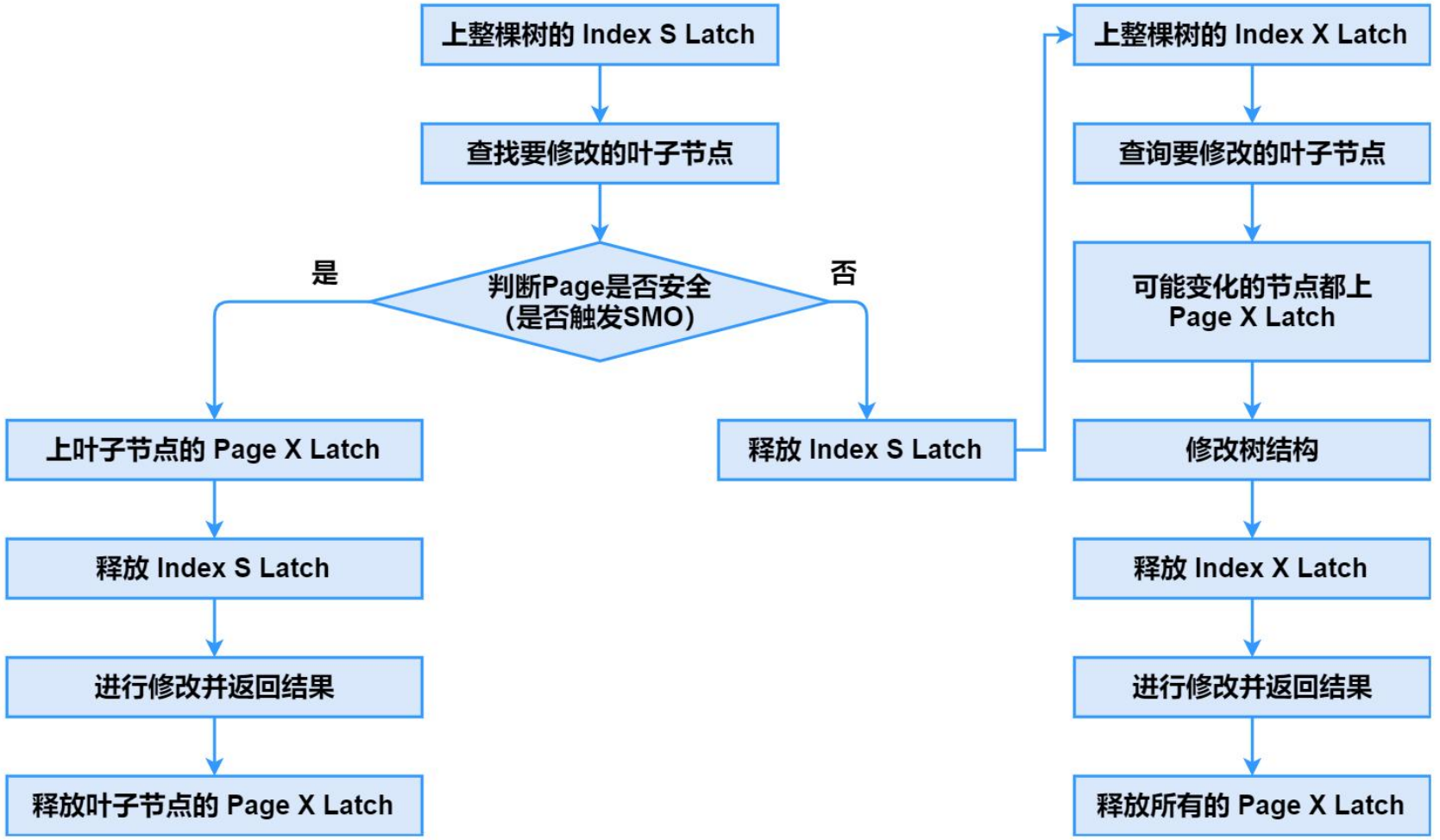
Latch 是我们传统意义上的锁，可以称为闩锁，保护内存中的物理数据。

在多线程场景下，内存池中的一个 B-tree 节点在被一个线程读取时，不能被另一个线程修改，这种场景就是多线程编程中共享数据的临界区问题。数据库中使用 latch 来控制单个 B-tree 节点的访问，从而保持 B-tree 物理结构的一致性，通常在每个节点的描述符中嵌入一个对应的 latch。

Latch Coupling：当从B+Tree的一个父节点到子节点，这期间不能有其他线程改变子节点，这时候需要持有父节点的 Latch 直到查询到子节点的 Latch。

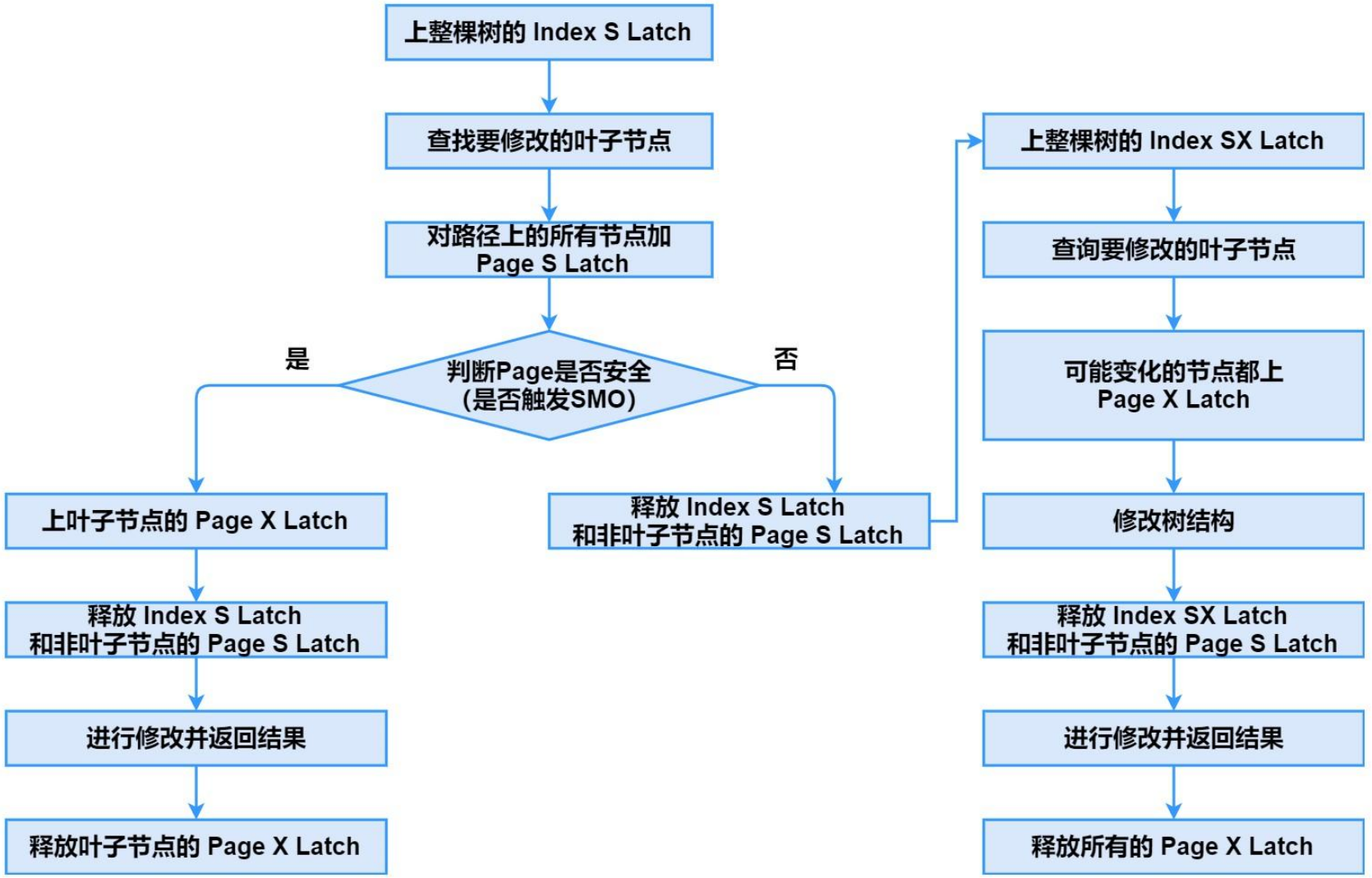
4. 校验锁和加锁-Latch

MySQL 5.6及之前版本写操作在触发SMO操作的情况下，因为持有Index X Latch，所有操作都无法进行。



4. 校验锁和加锁-Latch

MySQL 5.7及之后的8.0版本，针对SMO操作阻塞读的问题，引入例如 SX Latch，和 X Latch 和 SX Latch 冲突，但是和 S Latch 不冲突



4. 校验锁和加锁-Latch

B+树的问题在于，自上而下的搜索过程决定了加锁过程也是自上而下的。

哪怕是对一个个小小的叶子结点做读写操作，也要对根节点上Latch。一旦触发了SMO操作，那么好了，整棵树都不能动了。

后来， B树出现了其他变种，B*树、B-Link树、COW B树、Bw树等等，支持更强的并发能力。

还有，今天追加写的LSM树，Facebook 的 MyRocks存储引擎就是基于RocksDB（LevelDB）研发的。

4. 校验锁和加锁-Lock

InnoDB 的两阶段加锁将 Lock 的申请和释放分为两步：

1. 在事务过程中统一加锁；
2. 在事务提交或者回滚后统一放锁。

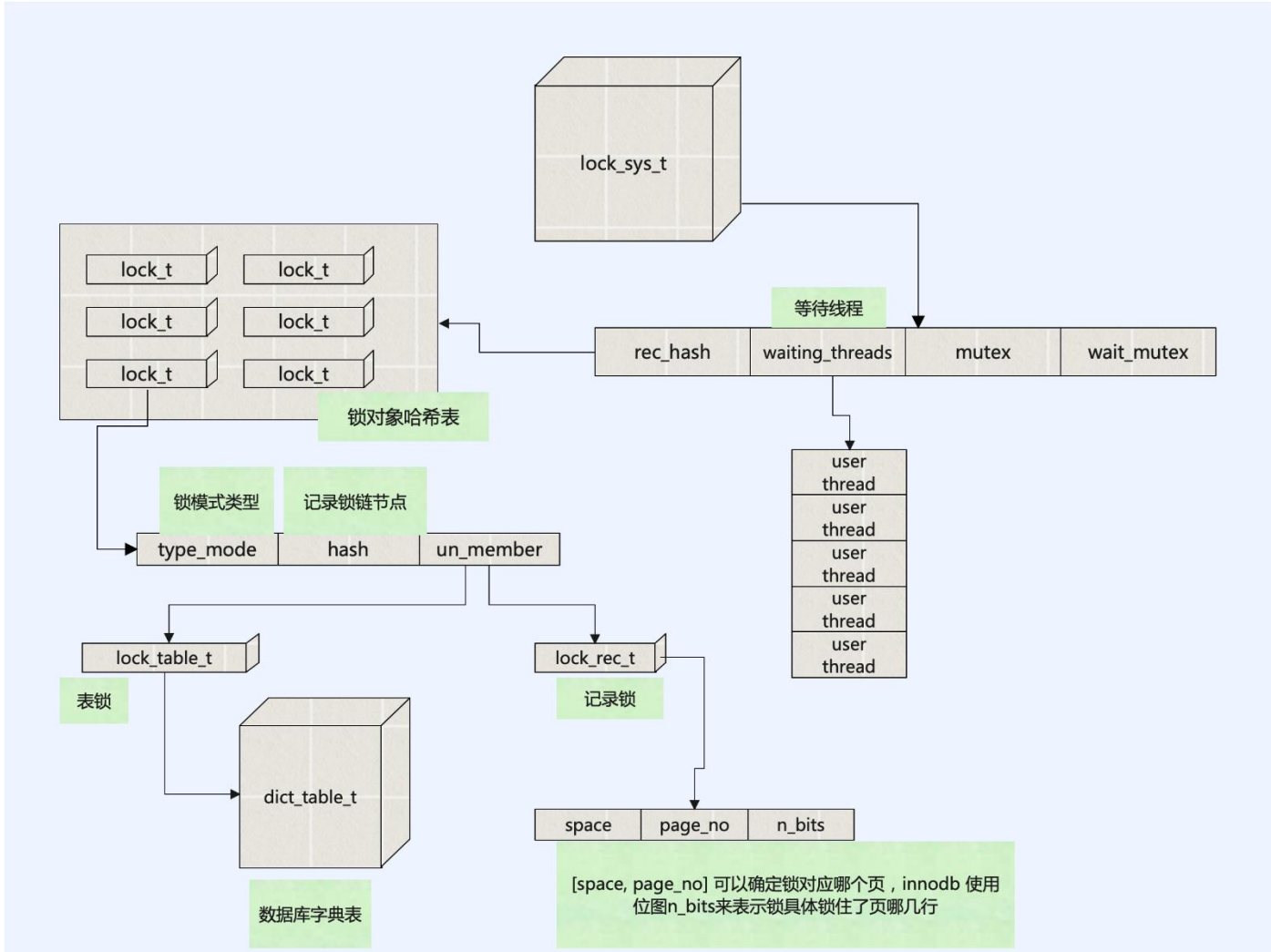
事务在创建 Lock 对象的过程中，需要判断是否与其他事务持有的 Lock 冲突。

对于冲突情况，需要进入 waiting 队列，而在持有 Lock 的事务提交或者回滚释放锁之后，选择等待队列中事务进行 Grant Lock。

4. 校验锁和加锁-Lock

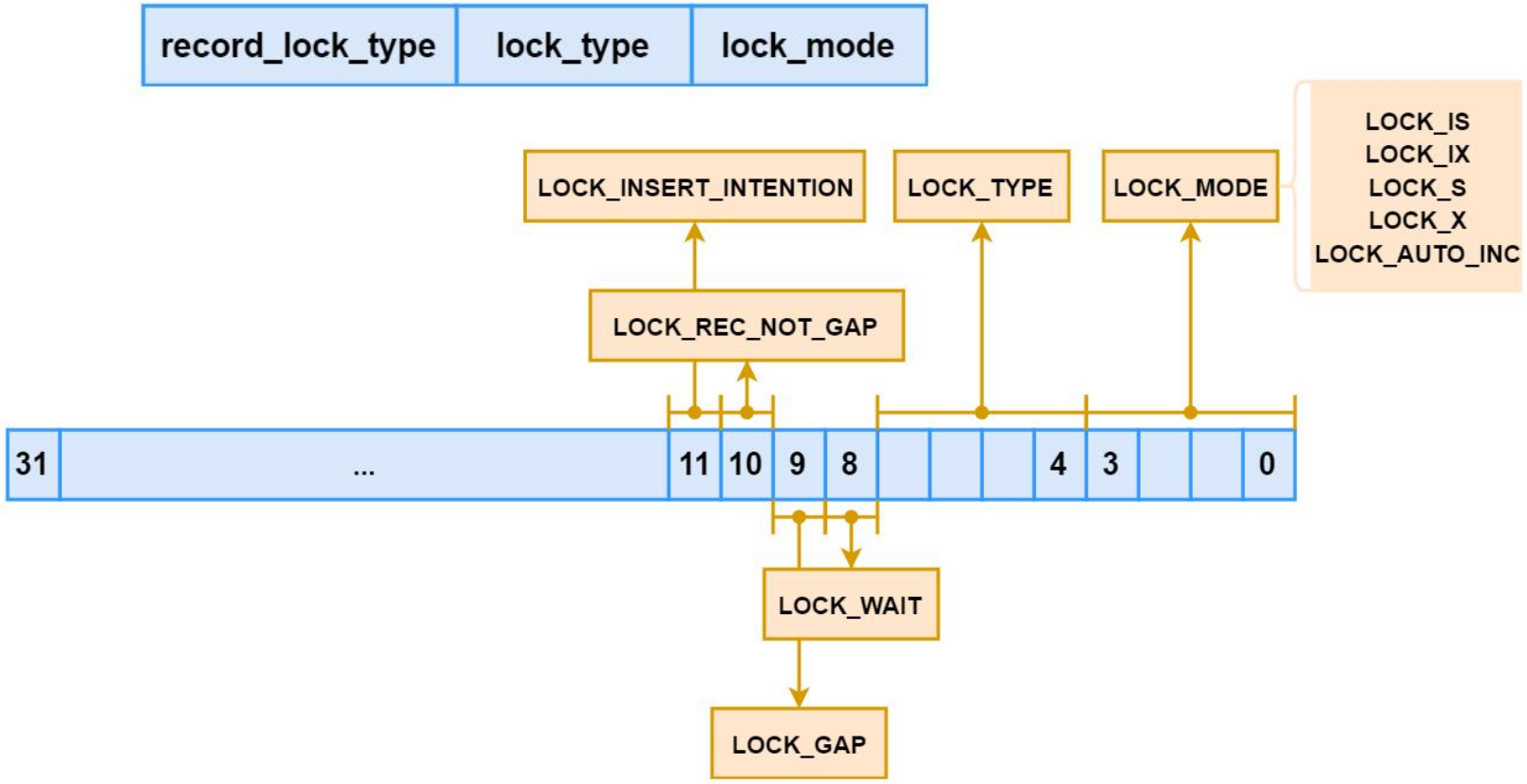
Lock的静态数据结构 lock_t

trx 事务
trx_locks 事务持有的锁链表
type_mode 锁类型和模式
hash 记录锁hash链节点
index 记录锁索引
un_member 表锁/行锁信息



4. 校验锁和加锁-Lock

InnoDB 使用 int32 存储 Lock 的 type mode。



0-3：表示锁模式

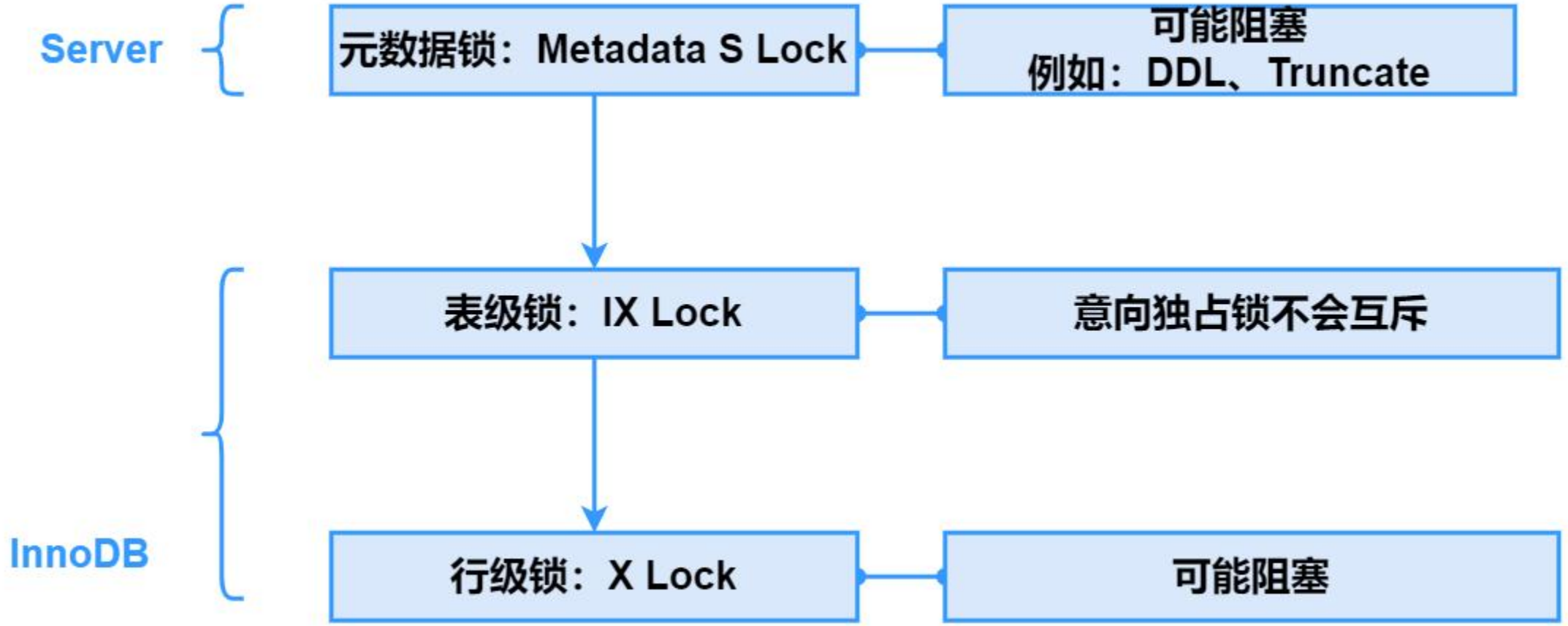
4：表示是否表锁

5：表锁是否行锁

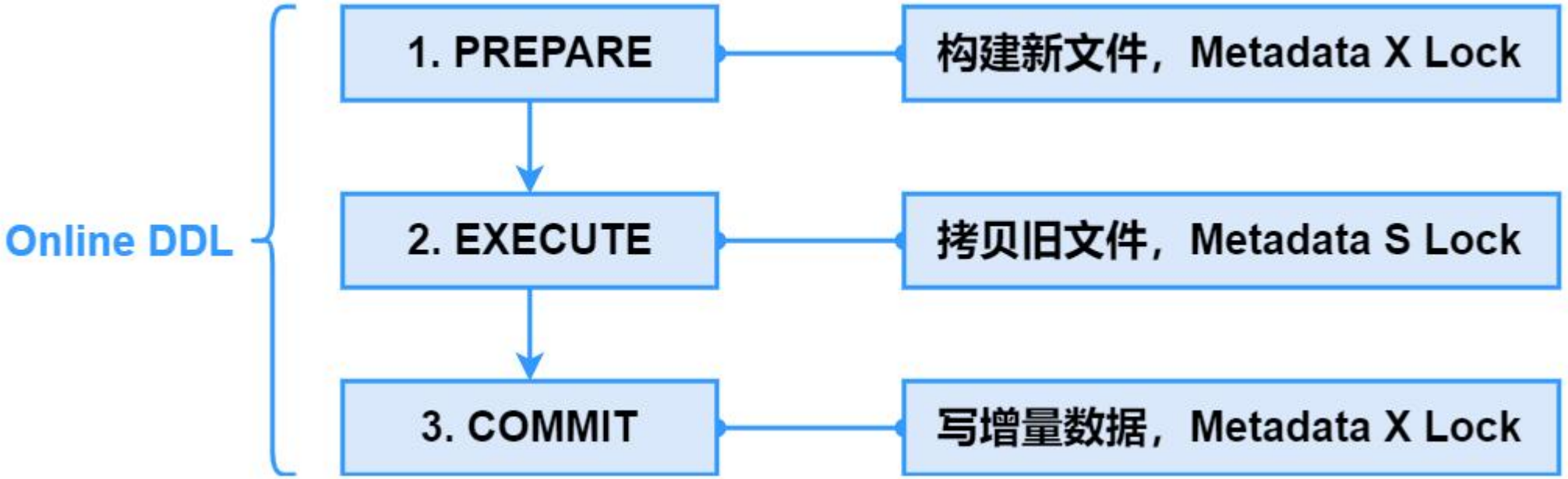
8：表示是否锁等待

9-31：表示记录锁类型

4. 校验锁和加锁-Lock



4. 校验锁和加锁-Lock



- Online DDL 有一个锁降级和升级的过程。
- 在 PREPPARE 和 COMMIT 阶段持有元数据写锁，会阻塞写操作。
- 在 EXECUTE 阶段降级为元数据读锁，写操作可以正常进行。

5. 修改数据、生成日志

加锁之后，就可以安全地对数据进行修改操作。

InnoDB 存储引擎在这一步主要写三部分内容：

1. 数据页：修改数据的本体；
2. undo log：MEM 日志，实现 MVCC 快照隔离，以及事务回滚；
3. redo log：WAL 日志，保证数据页、undo log 的安全，用于崩溃恢复。

5. 修改数据、生成日志

（1）数据页

- 修改前后这行数据的大小完全没变：就地更新。
- 任何字段的大小发生了变化：先删后插。

主要流程：

- 修改数据
- 数据页进入 buffer pool 的 flush list
- 释放 Page X Latch

如果修改超过数据页的空间上限，会触发页的分裂，会导致主键索引B+树的一系列SMO操作（Structure Modification Operation），这里不做详细讨论。

5. 修改数据、生成日志

（2）undo log

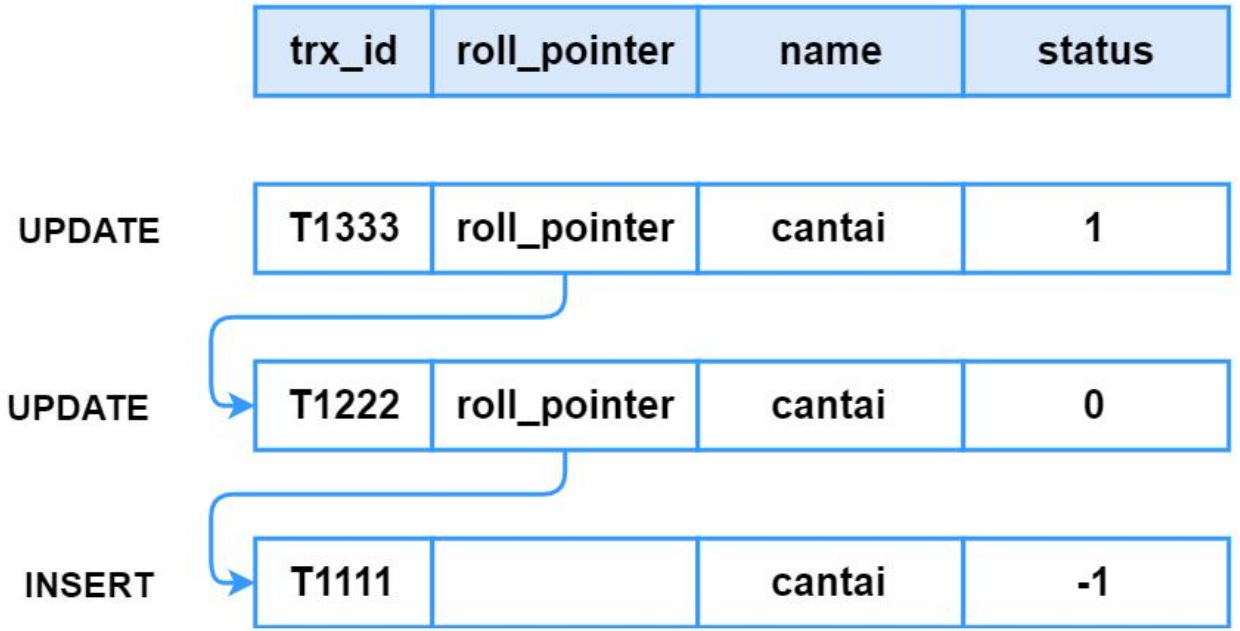
undo log 记录的是事务T修改前的值X和修改后的值Y，形成一个<T, X, Y>三元组。

undo log有两种格式：

5. 修改数据、生成日志

（2）undo log

MVCC的实现：查询语句在查询前生成一个 ReadView，和查找到数据的 trx_id 比较，不符合条件则通过 roll_pointer 向前追溯，直至找到符合条件的版本。

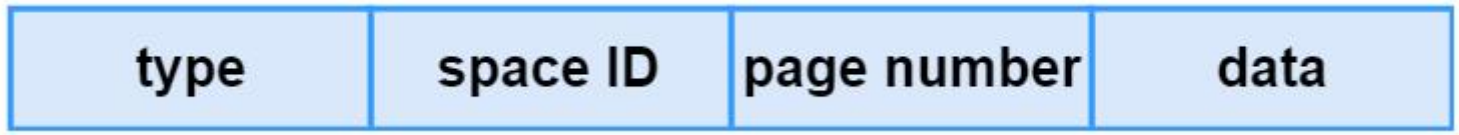


5. 修改数据、生成日志

（3）redo log

redo log 作为 wal 日志，为了保证 crash-safe，需要记录的内容非常多。

redo log 格式如下所示：



type：redo log 的类型（MySQL8.0中，有65种）

space ID：表空间号

page number: 页号

data: 把页中哪个位置修改成了哪些值

5. 修改数据、生成日志

（3）redo log

在当前例子中可能需要记录的 redo log 有：

1. 因为这行数据的字符数发生了变化，要删除旧记录，再插入新记录；
2. 要对上一条记录的 next_record 属性进行修改；
3. 数据页的 Page Directory 和 Page Header 的内容变化；
4. 导致节点分裂与合并的情况。

5. 修改数据、生成日志

（3）redo log

MTR(Mini-Transaction) 概念

一个MTR可以包含一组 redo log，无论是写入还是恢复时，都需要保证这组 redo log 的原子性。

一个 redo log 的最后，会有一种特殊 type 的 redo log 被生成并写入；

恢复时，只有读取到这个特殊 type 的 redo log，才认为这个 redo log 组是完整的。

5. 修改数据、生成日志

（3）redo log

redo log有全局递增序列号 LSN，在生成时需要写入 log buffer。

log buffer 是一块连续内存空间，由一个个 512B 的 block 组成。

- redo log block: 512B
- log buffer: 16MB (innodb_log_buffer_size参数控制)

1. buf_next_to_write：标记已经刷盘的 redo log 的位置
2. buf_free：记录最新生成 redo log 的位置

5. 修改数据、生成日志

（3）redo log

SQL 执行完成，事务提交前，redo log 是否需要落盘?

是不需要的，对于整个SQL执行过程中产生的任何信息：修改的数据页、redo log、undo log、binlog，当前都仅存在于内存中，即时宕机丢失，也没有任何问题，相当于这条 SQL 从未执行过。

但是还是有很多可能原因导致 redo log 落盘：

1. 事务提交时；
2. log buffer 空间不足（低于50%）时；
3. 后台线程周期性刷 log buffer；
4. MySQL 服务正常关闭；
5. write pos 超过 checkpoint 时。

6. 本地提交

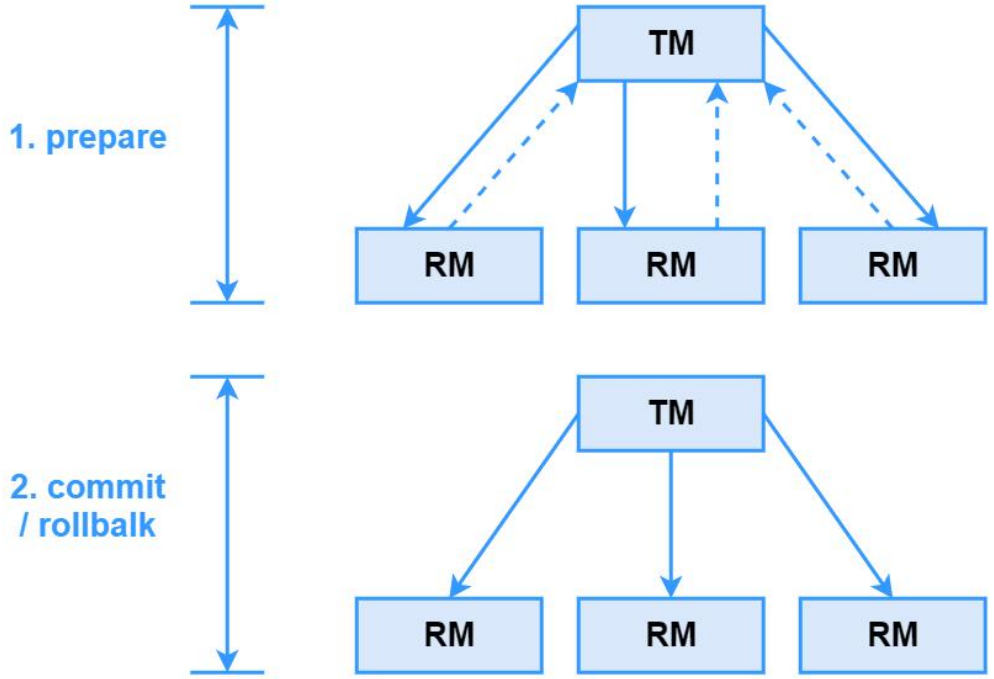
提交阶段 InnoDB 存储引擎 要写 redo log，MySQL 服务器要写 binlog。

Q：如何保证 redo log 和 binlog 的状态一致性?

6. 本地提交

分布式事务 XA （eXtended Architecture）协议，属于二阶段提交（2PC）。

XA协议中，分为两个角色：事务管理器（ TM ）和 资源管理器 （ RM ）。



6. 本地提交

MySQL 本地提交使用的 XA 协议，binlog 作为 TM，redo log 作为 RM。

1. prepare 阶段：InnoDB write/sync redo log；

⇒ TRX_PREPARED

2. commit 阶段：MySQL Server write/sync binlog，InnoDB commit。

⇒ TRX_NOT_STRATED

宕机时不同状态的处理：

- 事务状态为 TRX_ACTIVE，直接回滚事务；
- 事务状态为 TRX_NOT_STARTED，表示事务的 redo log 和 binlog 均已落盘，事务已提交；
- 事务状态为 TRX_PERPARED，根据 binlog 的写入状态来判断提交还是回滚，binlog 未写入成功则回滚，binlog 写入成功则提交并修改事务状态为 TRX_NOT_STARTED。

6. 本地提交

刷盘时机

对于 binlog，控制参数是 sync_binlog：

- 0：关闭写 binlog
- 1：每次提交都刷盘（默认）
- N：binlog 组提交

对于 log buffer，控制参数是 innodb_flush_log_at_trx_commit：

- 0：每秒刷盘（可能丢失事务）
- 1：每次提交写刷盘（默认）
- 2：每秒或者每次提交刷盘

7. 主备复制

主备复制的策略：

- 异步复制：主库写完 binlog 后即可返回提交成功，无需等待备库响应。
- 半同步复制：主库接收到指定数量的备机转储 relay log 成功的 ACK 后，返回提交成功。
- 同步复制：主库等到备库回放 relay log 执行完事务后才可提交成功。

Q：备库一直没有响应怎么办？

A：MySQL 原生的半同步复制机制在这里会有一个超时时间，超过这个时间备库还没有响应，主机自动提交。这里是不安全的，因为半同步复制退化成了异步复制。

8. 脏页刷盘

MySQL 的页大小默认是 16K，Linux 的页大小默认是4K，因此 MySQL的一页数据需要分四次刷盘上，因此这个操作并非原子的。

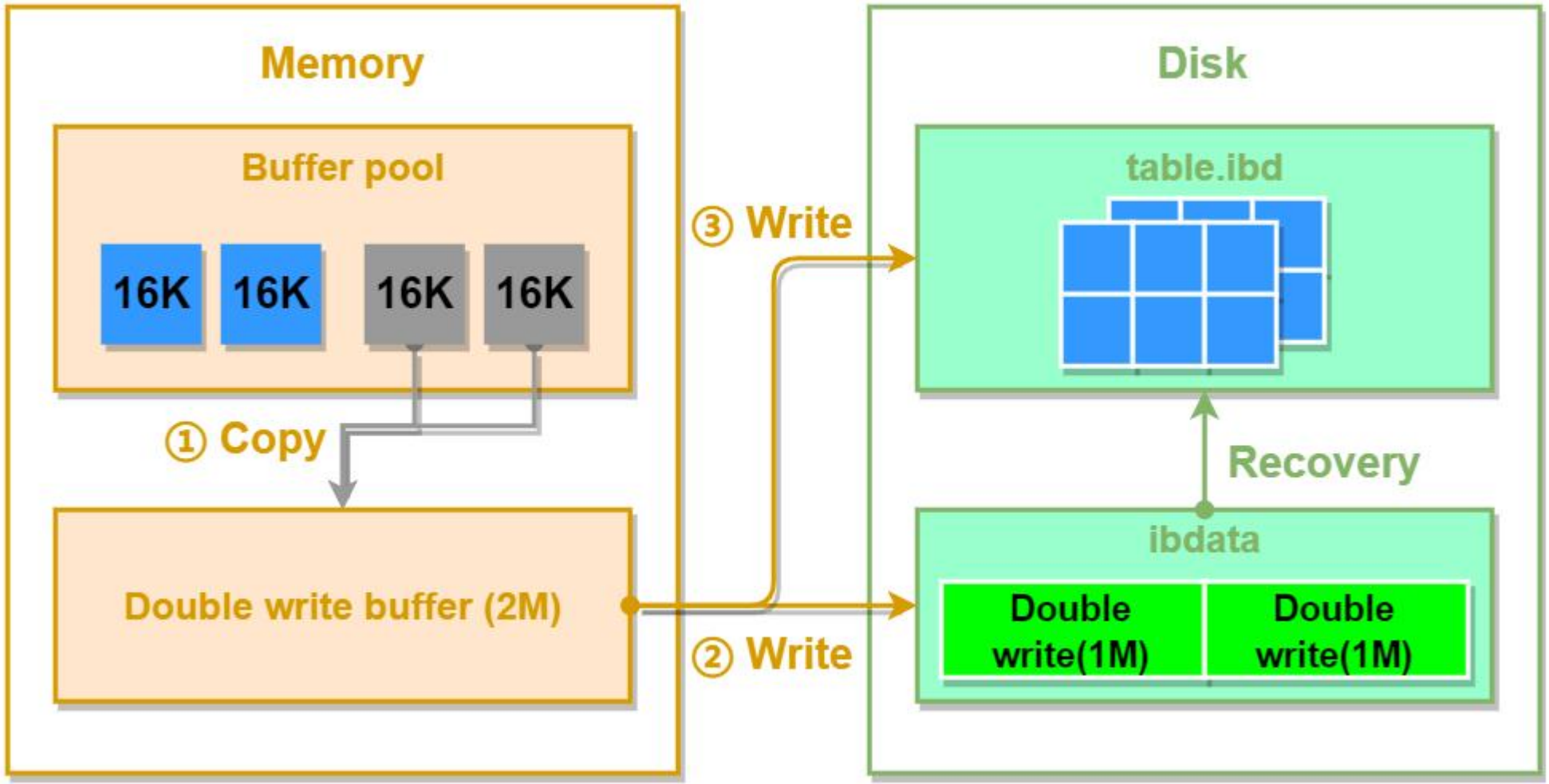
比如 OS 在写第二个页的时候断电，这时候会造成页的数据损坏，这种损坏依靠 redo log 是无法恢复的（redo log 记录是对页是物流操作，不会记录页的全量数据）。

InnoDB 使用 Doublewrite Buffer 解决这个问题，虽然名字中带有 buffer，但是它其实是内存+磁盘双重结构。

当有数据刷盘时：

1. 页数据先通过 memcpy 函数拷贝至内存中的 Doublewrite Buffer中；
2. Doublewrite Buffer 内存中的数据页，通过 fsync 刷到 Doublewrite Buffer 的磁盘上，分两次写入磁盘共享表空间，每次写1MB（顺序写，性能很高）；
3. Doublewrite Buffer 内存中的数据页，再刷到数据磁盘存储的 ibd 文件中（离散写）。

8. 脏页刷盘


Memory
Buffer pool
16K
16K
16K
16K
① Copy
Double write buffer (2M)
③ Write
② Write
Disk
table.ibd
Recovery
ibdata
Double write(1M)
Double write(1M)

Thanks