数据库内核月报 － 2019 / 03

当期文章

PgSQL · 特性分析 · 内存管理机制

背景

为了提高数据访问的速度，一般数据库操作系统都会引入内存作为缓存，而为了方便管理和合并I/O，一般会开辟一个缓存池（buffer pool）。本文主要讲述PostgreSQL 如何进行缓存池管理。

数据库物理结构

在下文讲述缓存池管理之前，我们需要简单介绍下PostgreSQL 的数据库集簇的物理结构。PostgreSQL 的数据文件是按特定的目录和文件名组成：

- 如果是特定的tablespace 的表/索引数据，则文件名形如

```
$PGDATA/pg_tblspc/$tablespace_oid/$database_oid/$relation_oid.n
```

- 如果不是特定的tablespace 的表/索引数据，则文件名形如

```
$PGDATA/base/$database_oid/$relation_oid.num
```

其中PGDATA 是初始化的数据根目录，tablespace_oid 是tablespace 的oid，database_oid 是database 的oid，relation_oid是表/索引的oid。no 是一个数值，当表/索引的大小超过了1G（该值可以在编译PostgreSQL 源码时由configuration的–with-segsize 参数指定大小），该数值就会加1，初始值为0，但是为0时文件的后缀不加.0。

除此之外，表/索引数据文件中还包含以_fsm（与free space map 相关，详见文档 ） 和 _vm （与visibility map 相关，详见文档 ）为后缀的文件。这两种文件在PostgreSQL 中被认为是表/索引数据文件的另外两种副本，其中_fsm 结尾的文件为该表/索引的数据文件副

本1，_vm结尾的文件为该表/索引的数据文件副本2，而不带这两种后缀的文件为该表/索引的数据文件副本0。

无论表/索引的数据文件副本0或者1或者2，都是按照页面（page）为组织单元存储的，具体数据页的内容和结构，我们这里不再详细展开。但是值得一提的是，缓冲池中最终存储的就是一个个的page。而每个page我们可以按照(tablespace_oid, database_oid, relation_oid, fork_no, page_no) 唯一标示，而在PostgreSQL 源码中是使用结构体 BufferTag 来表示，其结构如下，下文将会详细分析这个唯一标示在内存管理中起到的作用。

```
typedef struct buftag
{
        RelFileNode rnode;                      /* physical relati
        ForkNumber      forkNum;
        BlockNumber blockNum;           /* blknum relative to begi
} BufferTag;
typedef struct RelFileNode
{
    Oid         spcNode;        /* tablespace */
    Oid         dbNode;         /* database */
    Oid         relNode;        /* relation */
} RelFileNode;
```

缓存管理结构

在PostgreSQL中，缓存池可以简单理解为在共享内存上分配的一个数组，其初始化的过程如下:

```
BufferBlocks = (char *) ShmemInitStruct("Buffer Blocks", NBuffers
```

其中NBuffers 即缓存池的大小与GUC 参数shared_buffers 相关（详见链接）。数组中每个元素存储一个缓存页，对应的下标buf_id 可以唯一标示一个缓存页。

为了对每个缓存页进行管理，我们需要管理其元数据，在PostgreSQL 中利用BufferDesc 结构体来表示每个缓存页的元数据，下文称其为缓存描述符，其初始化过程如下：

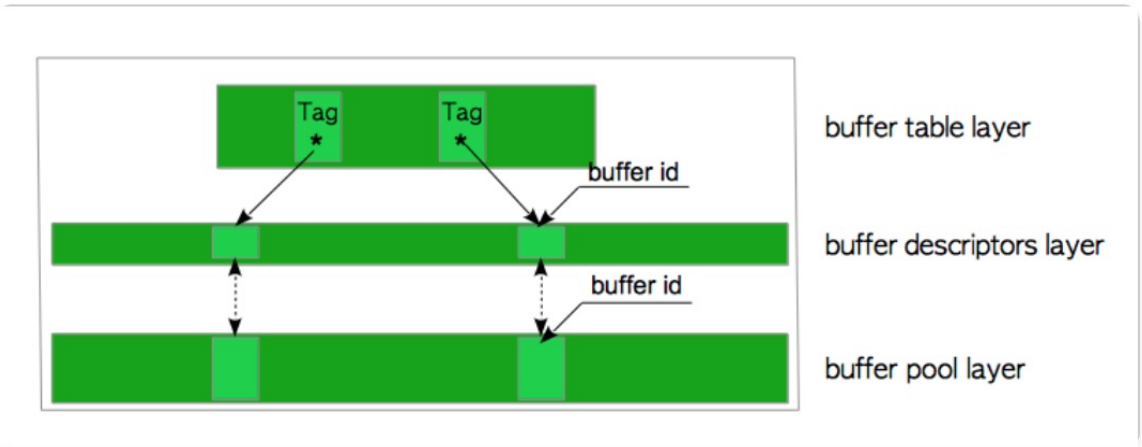
```
BufferDescriptors = (BufferDescPadded *)
                                    ShmemInitStruct("Buffer Descriptor
                                                                    NBuf
                                                                     &fou
```

可以发现，缓存描述符是和缓存池的每个页面一一对应的，即如果有16384 个缓存页面，则就有16384 个缓存描述符。而其中的BufferTag 即是上文的PostgreSQL 中数据页面的唯一标示。

直到这里，我们如果要从缓存池中请求某个特定的页面，只需要遍历所有的缓存描述符即可。但是很显然这样的性能会非常的差。为了优化这个过程，PostgreSQL 引入了一个BufferTag 和缓存描述符的hash 映射表。通过它，我们可以快速找到特定的数据页面在缓存池中的位置。

概括起来，PostgreSQL 的缓存管理主要包括三层结构，如下图：

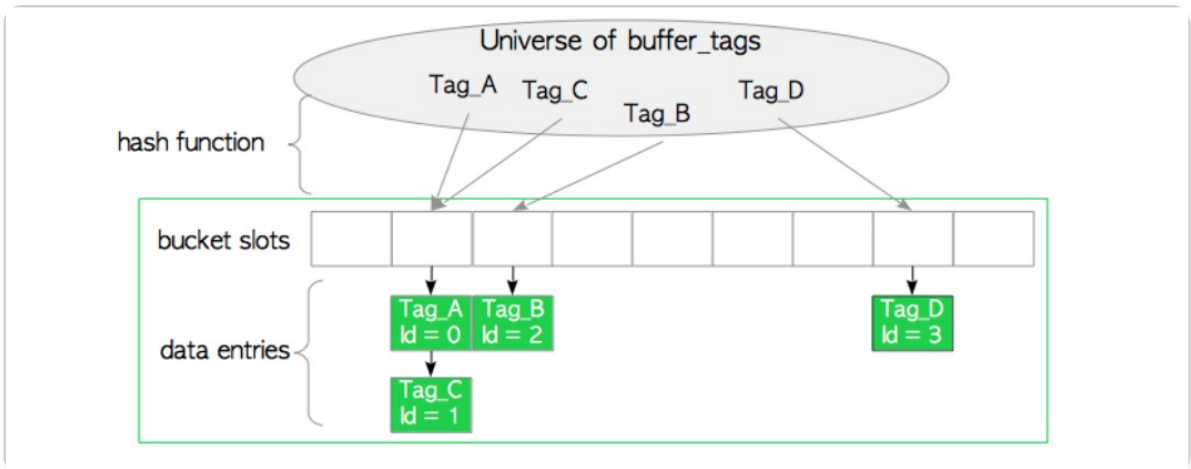
- 缓存池，是一个数组，每个元素其实就是一个缓存页，下标buf_id 唯一标示一个缓存页。
- 缓存描述符，也是一个数组，而且和缓存池的缓存一一对应，保存每个缓存页的元数据信息。
- 缓存hash 表，是存储BufferTag 和缓存描述符之间映射关系的hash 表。



下文，我们将分析每层结构的具体实现以及涉及到的锁管理和缓冲页淘汰算法，深入浅出，介绍PostgreSQL 缓存池的管理机制。

缓存hash 表

从上文的分析，我们知道缓存hash 表是为了加快BufferTag 和缓存描述符的检索速度构造的数据结构。在PostgreSQL 中，缓存hash 表的数据结构设计比较复杂，我们会在接下来的月报去介绍在PostgreSQL 中缓存hash 表是如何实现的。在本文中，我们把缓存hash 表抽象成一个个的bucket slot。因为哈希函数还是有可能碰撞的，所以bucket slot 内可能有几个data entry 以链表的形式存储，如下图：



而使用BufferTag 查找对应缓存描述符的过程可以简述如下：

- 获取BufferTag 对应的哈希值hashvalue
- 通过hashvalue 定位到具体的bucket slot
- 遍历bucket slot 找到具体的data entry，其数据结构BufferLookupEnt 如下：

```
/* entry for buffer lookup hashtable */
typedef struct
{
        BufferTag       key;                    /* Tag of a disk p
        int                     id;                             /*
} BufferLookupEnt;
```

BufferLookupEnt 的结构包含id 属性，而这个属性可以和唯一的缓存描述符或者唯一的缓存页对应，所以我们就获取了BufferTag 对应的缓存描述符。

缓存hash 表初始化的过程如下：

```
InitBufTable(NBuffers + NUM_BUFFER_PARTITIONS);
```

可以看出，缓存hash 表的bucket slot 个数要比缓存池个数NBuffers 要大。除此之外，多个后端进程同时访问相同的bucket slot 需要使用锁来进行保护，后文的锁管理会详细讲述这个过程。

缓存描述符

上文的缓存hash 表可以通过BufferTag 查询到对应的 buffer ID，而PostgreSQL 在初始化缓存描述符和缓存页面一一对应，存储每个缓存页面的元数据信息，其数据结构BufferDesc 如下：

```
typedef struct BufferDesc
{
        BufferTag       tag;                    /* ID of page cont
        int                     buf_id;                 /* buffer

        /* state of the tag, containing flags, refcount and usagec
        pg_atomic_uint32 state;

        int                     wait_backend_pid;       /* backenc
        int                     freeNext;               /* link ir

        LWLock          content_lock;   /* to lock access to buffe
} BufferDesc;
```

其中：

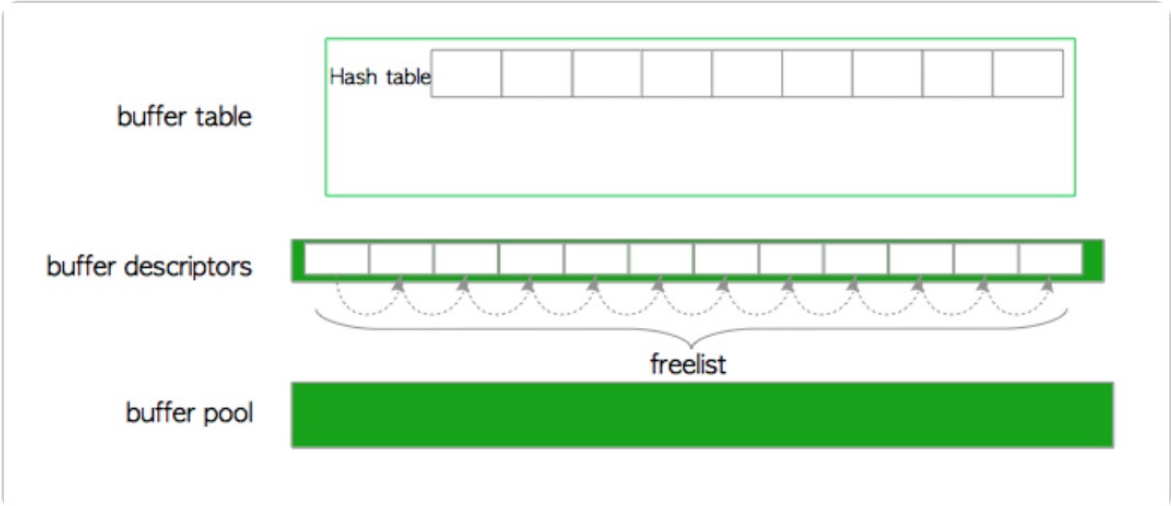
- tag 指的是对应缓存页存储的数据页的唯一标示

- buffer_id 指的是对应缓存页的下标，我们通过它可以直接访问对应缓存页
- state 是一个无符号32位的变量，包含：
  - 18 bits refcount，当前一共有多少个后台进程正在访问该缓存页，如果没有进程访问该页面，本文称为该缓存描述符unpinned，否则称为该缓存描述符pinned
  - 4 bits usage count，最近一共有多少个后台进程访问过该缓存页，这个属性用于缓存页淘汰算法，下文将具体讲解。
  - 10 bits of flags，表示一些缓存页的其他状态，如下：

```
#define BM_LOCKED                               (1U << 22)      /*
#define BM_DIRTY                                (1U << 23)      /*
#define BM_VALID                                (1U << 24)      /*
#define BM_TAG_VALID                    (1U << 25)      /* tag is
#define BM_IO_IN_PROGRESS               (1U << 26)      /* read or
#define BM_IO_ERROR                             (1U << 27)      /*
#define BM_JUST_DIRTIED                 (1U << 28)      /* dirtied
#define BM_PIN_COUNT_WAITER             (1U << 29)      /* have wa
#define BM_CHECKPOINT_NEEDED    (1U << 30)      /* must write for
#define BM_PERMANENT                    (1U << 31)      /* permane
```

- freeNext，指向该缓存之后第一个空闲的缓存描述符
- content_lock，是控制缓存描述符的一个轻量级锁，我们会在缓存锁管理具体分析其作用

上文讲到，当数据启动时，会初始化与缓存池大小相同的缓存描述符数组，其每个缓存描述符都是空的，这时整个缓存管理的三层结构如下：



第一个数据页面从磁盘加载到缓存池的过程可以简述如下：

- 从freelist 中找到第一个缓存描述符，并且把该缓存描述符pinned （增加refcount和usage_count）
- 在缓存hash 表中插入这个数据页面的BufferTag 与buf_id 的对应新的data entry
- 从磁盘中将数据页面加载到缓存池中对应缓存页面中
- 在对应缓存描述符中更新该页面的元数据信息

缓存描述符是可以持续更新的，但是如下场景会使得对应的缓存描述符状态置为空并且放在freelist 中：

- 数据页面对应的表或者索引被删除
- 数据页面对应的数据库被删除
- 数据页面被VACUUM FULL 命令清理

缓存池

上文也提到过，缓存池可以简单理解为在共享内存上分配的一个缓存页数组，每个缓存页大小为PostgreSQL 页面的大小，一般为8KB，而下标buf_id 唯一标示一个缓存页。

缓冲池的大小与GUC 参数shared_buffers 相关，例如shared_buffers 设置为128MB，页面大小为8KB，则有128MB/8KB=16384个缓存页。

缓存锁管理

在PostgreSQL 中是支持并发查询和写入的，多个进程对缓存的访问和更新是使用锁的机制来实现的，接下来我们分析下在PostgreSQL 中缓存相关锁的实现。

因为在缓存管理的三层结构中，每层都有并发读写的情况，通过控制缓存描述符的并发访问就能够解决缓存池的并发访问，所以这里的缓存锁实际上就是讲的缓存hash 表和缓存描述符的锁。

BufMappingLock

BufMappingLock 是缓存hash 表的轻量级锁。为了减少BufMappingLock 的锁争抢并且能够兼顾锁空间的开销，PostgreSQL 中把BufMappingLock 锁分为了很多片，默认为128片，每一片对应总数/128 个bucket slot。

当我们检索一个BufferTag 对应的data entry是需要BufMappingLock 对应分区的共享锁，当我们插入或者删除一个data entry 的时候需要BufMappingLock 对应分区的排他锁。

除此之外，缓存hash 表还需要其他的一些原子锁来保证一些属性的一致性，这里不再赘述。

content_lock

content_lock 是缓存描述符的轻量级锁。当需要读一个缓存页的时候，后台进程会去请求该缓存页对应缓存描述符的content_lock 共享锁。而当以下的场景，后台进程会去请求content_lock 排他锁：

- 插入或者删除/更新该缓存页的元组
- vacuum 该缓存页
- freeze 该缓存页

io_in_progress_lock

io_in_progress_lock 是作用于缓存描述符上的I/O锁。当后台进程将对应缓存页加载至缓存或者刷出缓存，都需要这个缓存描述符上的io_in_progress_lock 排它锁。

其他

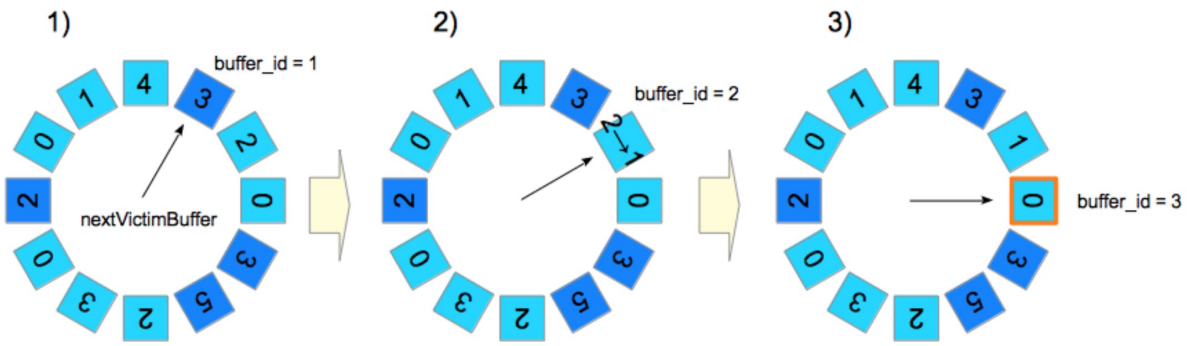
缓存描述符中的state 属性含有很多需要原子排他性的字段，例如refcount 和 usage count。但是在这里没有使用锁，而是使用pg_atomic_unlocked_write_u32()或者pg_atomic_read_u32() 方法来保证多进程访问相同缓存描述符的state 的原子性。

缓存页淘汰算法

在PostgreSQL 中采用clock-sweep 算法来进行缓存页的淘汰。clock-sweep 是NFU（Not Frequently Used）最近不常使用算法的一个优化算法。其算法在PostgreSQL 中的实现可以简述如下：

1. 获取第一个候选缓存描述符，存储在freelist 控制信息的数据结构BufferStrategyControl的nextVictimBuffer 属性中
2. 如果该缓存描述符unpinned，则跳到步骤3，否则跳到步骤4
3. 如果该候选缓存描述符的usage_count 属性为0，则选取该缓存描述符为要淘汰的缓存描述符，跳到步骤5，否则，usage_count–，跳到步骤4
4. nextVictimBuffer 赋值为下一个缓存描述符（当缓存描述符全部遍历完成，则从第0个继续），跳到步骤1继续执行，直到发现一个要淘汰的缓存描述符
5. 返回要淘汰的缓存描述符的buf_id

clock-sweep 算法一个比较简单的例子如下图：



总结

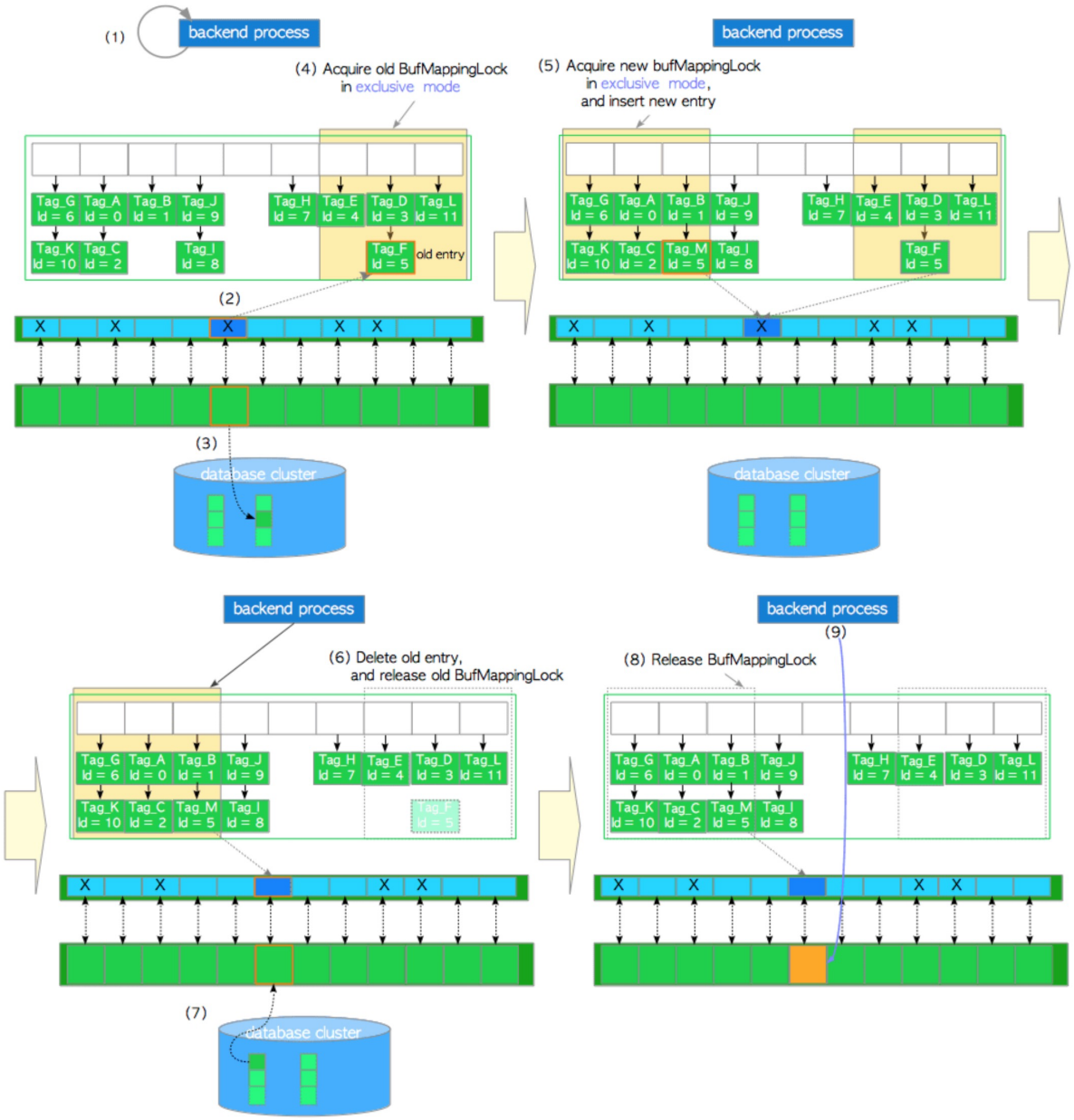
至此，我们已经对PostgreSQL 的缓存池管理整个构架和一些关键的技术有了了解，下面我们会举例说明整个流程。

为了能够涉及到较多的操作，我们将**缓存池满后访问某个不在缓存池的数据页面**这种场景作为例子，其整个流程如下：

1. 根据请求的数据页面形成BufferTag，假设为Tag_M，用Tag_M 去从缓存hash 表中检索data entry，很明显这里没有发现该BufferTag
2. 使用clock-sweep 算法选择一个要淘汰的缓存页面，例如这里buf_id=5，该缓存页的data entry 为’Tag_F, buf_id=5’
3. 如果是脏页，将buf_id=5的缓存页刷新到磁盘，否则跳到步骤4。刷新一个脏页的步骤如下：

   a. 获得buffer_id=5的缓存描述符的content_lock 共享锁和io_in_progress 排它锁（步骤f会释放） b. 修改该描述符的state，BM_IO_IN_PROGRESS 和BM_JUST_DIRTIED 字段设为1 c. 根据情况，执行 XLogFlush() 函数，对应的wal 日志刷新到磁盘 d. 将缓存页刷新到磁盘 e. 修改该描述符的state，BM_IO_IN_PROGRESS 字段设为1，BM_VALID 字段设为1 f. 释放该缓存描述符的content_lock 共享锁和io_in_progress 排它锁
4. 获取buf_id=5的bucket slot 对应的BufMappingLock 分区排他锁，并将该data entry 标记为旧的
5. 获取新的Tag_M 对应bucket slot 的BufMappingLock 分区排他锁并且插入一条新的data entry
6. 删除buf_id=5的data entry，并且释放buf_id=5的bucket slot 对应的BufMappingLock 分区锁
7. 从磁盘上加载数据页面到buf_id=5的缓存页面，并且更新buf_id=5的缓存描述符state属性的BM_DIRTY字段为0，初始化state的其他字段。
8. 释放Tag_M 对应bucket slot 的BufMappingLock 分区排他锁
9. 从缓存中访问该数据页面

其过程的示意图如下所示：



阿里云PolarDB-数据库内核组

社招/校招 欢迎投递简历 直达招聘信息

直连月报小编

欢迎在github上star AliSQL

阅读： -