Postgres 学习手册： 原理

jask

09/26/2024

缓冲区管理器

PostgreSQL 缓冲区管理器由缓冲表、缓冲区描述符和缓冲池组成

缓冲池是一个数组，数据的每个槽中存储数据文件的一页。缓冲池数组的序号索引称为 buffer_id。

PostgreSQL 中的每个数据文件页面都可以分配到唯一的标签，即缓冲区标签。当缓冲区管理器收到请求时，PostgreSQL 会用到目标页面的缓冲区标签。

缓冲区标签由三个值组成，分别是关系文件节点、关系分支编号和页面块号。

第一个值分别代表了表空间、数据库和表的 oid；第二个值代表关系表的分支号；最后一个值代表页面号。那么该标签表示，在某个表空间（oid=16821）中，某个数据库（oid=16384）的某张表（oid=37721）的 0 号分支（0 代表关系表本体）的第 7 号页面。再比如，缓冲区标签 {(16821,16384,37721),1,3} 表示该表空闲空间映射文件的三号页面。关系本体 main 分支编号为 0，空闲空间映射 fsm 分支编号为 1。

缓冲区管理器的结构

缓冲表层是一个散列表，它存储着页面的 buffer_tag 与描述符的 buffer_id 之间的映射关系。

缓冲区描述符层是一个由缓冲区描述符组成的数组。每个描述符与缓冲池槽一一对应，并保存着相应槽的元数据。

缓冲池层是一个数组。每个槽都存储一个数据文件页，数组槽的索引称为 buffer_id。

缓冲表

缓冲表可以在逻辑上分为三个部分，分别是散列函数、散列桶槽及数据项

内置散列函数将 buffer_tag 映射到哈希桶槽。即使散列桶槽的数量比缓冲池槽的数量多，冲突也可能会发生。因此缓冲表采用了使用链表的分离链接方法来解决冲突。当数据项被映射至同一个桶槽时，该方法会将这些数据项保存在一个链表

数据项包括两个值，即页面的 buffer_tag 和包含页面元数据的描述符的 buffer_id。

缓冲区描述符

缓冲区描述符保存着页面的元数据，这些与缓冲区描述符相对应的页面保存在缓冲池槽中。缓冲区描述符的结构由 BufferDesc 结构定义。

polardb 版本

```
typedef struct BufferDesc
{
    BufferTag   tag;            /* ID of page contained in buffer */
    int         buf_id;         /* buffer's index number (from 0) */

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

    int         wait_backend_pid;   /* backend PID of pin-count waiter */
    int         freeNext;       /* link in freelist chain */

    LWLock      content_lock;   /* to lock access to buffer contents */

    /* POLAR */
```

```
    int         flush_next;      /* link to next dirty buffer */
    int         flush_prev;      /* link to prev dirty buffer */
    XLogRecPtr  oldest_lsn;      /* the first lsn which marked this buffer dirty */
    /*
     * If a buffer can not be flushed on primary because its latest modification
     * lsn > oldest apply lsn, in order to advance the consistent lsn, a copy
     * is made. So copy_buffer is used to point to its copied buffer of the
     * buffer.
     */
    CopyBufferDesc  *copy_buffer;
    uint8       polar_flags;
    uint16      recently_modified_count;
    /* POLAR: record buffer redo state */
    pg_atomic_uint32 polar_redo_state;
} BufferDesc;
```

tag 保存着目标页面的 buffer_tag，该页面存储在相应的缓冲池槽中

state 表示页面状态。

描述符层

缓冲区描述符的集合构成了一个数组，本书称该数组为缓冲区描述符层。

当 PostgreSQL 服务器启动时，所有缓冲区描述符的状态都为空。在 PostgreSQL 中，这些描述符构成了一个名为 freelist 的链表

(1) 从 freelist 的头部取一个空描述符，并将其钉住，即将 refcount 和 usage_count 增加 1。

(2) 在缓冲表中插入新项，该缓冲表项保存了页面 buffer_tag 与所获描述符 buffer_id 之间的关系。

(3) 将新页面从存储器加载至相应的缓冲池槽中。

(4) 将新页面的元数据保存至所获取的描述符中。

从 freelist 中摘出的描述符始终保存着页面的元数据。换言之，仍然在使用的非空描述符不会返还到 freelist 中。但当下列任一情况出现时，描述符状态将变为“空”，并被重新插入至 freelist 中。

1. 相关表或索引已被删除。

2. 相关数据库已被删除。

3. 相关表或索引已经被 VACUUM FULL 命令清理。

缓冲区管理器锁

缓冲表锁

BufMappingLock 保护整个缓冲表的数据完整性。它是一种轻量级的锁，有共享模式与独占模式。在缓冲表中查询条目时，后端进程会持有共享的 BufMappingLock。插入或删除条目时，后端进程会持有独占的 BufMappingLock。

BufMappingLock 会被分为多个分区，以减少缓冲表中的争用（默认为 128 个分区）。每个 BufMappingLock 分区都保护着一部分相应的散列桶槽。

缓冲表也需要许多其他锁。例如，在缓冲表内部会使用自旋锁（spin lock）来删除数据项。

描述符相关的锁

每个缓冲区描述符都会用到内容锁（content_lock）与 IO 进行锁（io_in_progress_lock）这两个轻量级锁，以控制对相应缓冲池槽页面的访问。当检查或更改描述符本身字段的值时，就会用到自旋锁。

内容锁　内容锁（content_lock）是一个典型的强制限制访问的锁，它有共享与独占两种模式。

当读取页面时，后端进程以共享模式获取页面相应缓冲区描述符中的 content_lock。

执行下列操作之一时，则会获取独占模式的 content_lock。

 将行（即元组）插入页面，或更改页面中元组的 t_xmin/t_xmax 字段时（简单地说，这些字段会在相关元组被删除或更新行时发生更改）。

 物理移除元组，或压紧页面上的空闲空间。

 冻结页面中的元组。

IO 进行锁 IO 进行锁（io_in_progress_lock）用于等待缓冲区上的 I/O 完成。当 PostgreSQL 进程加载/写入页面数据时，该进程在访问页面期间，持有对应描述符上独占的 io_in_progres_lock。

自选锁 当检查或更改标记字段与其他字段时，例如 refcount 和 usage_count，会用到自旋锁。下面是两个使用自旋锁的具体例子。

1. 钉住缓冲区描述符。

(1) 获取缓冲区描述符上的自旋锁。

(2) 将其 refcount 和 usage_count 的值增加 1。

(3) 释放自旋锁。

2. 将脏位设置为"1"。

(1) 获取缓冲区描述符上的自旋锁。

(2) 使用位操作将脏位置位为"1"。

(3) 释放自旋锁。

后来用原子操作替换了自旋锁。

缓冲区管理器的工作原理

访问存储在缓冲池中的页面

当从缓冲池槽中的页面里读取行时，PostgreSQL 进程获取相应缓冲区描述符的共享 content_lock，因而缓冲池槽可以同时被多个进程读取。

当向页面插入（及更新、删除）行时，该 postgres 后端进程获取相应缓冲区描述符的独占 content_lock（注意，这里必须将相应页面的脏位置设为"1"）。

访问完页面后，相应缓冲区描述符的引用计数值减 1。

我们来介绍最简单的情况，即所需页面已经存储在缓冲池中。在这种情况下，缓冲区管理器会执行以下步骤：

(1) 创建所需页面的 buffer_tag（在本例中 buffer_tag 是'Tag_C'），并使用散列函数计算与描述符相对应的散列桶槽。

(2) 获取相应散列桶槽分区上的 BufMappingLock 共享锁。

(3) 查找标签为'Tag_C'的条目，并从条目中获取 buffer_id。本例中 buffer_id 为 2。

(4) 将 buffer_id=2 的缓冲区描述符钉住，即将描述符的 refcount 和 usage_count 增加 1。

(5) 释放 BufMappingLock。

(6) 访问 buffer_id=2 的缓冲池槽。

将页面从存储加载到空槽

在第二种情况下，假设所需页面不在缓冲池中，且 freelist 中有空闲元素（空描述符）。这时，缓冲区管理器将执行以下步骤：

(1) 查找缓冲区表（本节假设页面不存在，找不到对应页面）。

第一，创建所需页面的 buffer_tag（本例中 buffer_tag 为'Tag_E'）并计算其散列桶槽。

第二，以共享模式获取相应分区上的 BufMappingLock。

第三，查找缓冲区表（根据假设，这里没找到）。

第四，释放 BufMappingLock。

(2) 从 freelist 中获取空缓冲区描述符，并将其钉住。在本例中所获的描述符：buffer_id=4。

(3) 以独占模式获取相应分区的 BufMappingLock（此锁将在步骤（6）中被释放）。

(4) 创建一条新的缓冲表数据项：buffer_tag='Tag_E',buffer_id=4，并将其插入缓冲区表中。(5) 将页面数据从存储加载至 buffer_id=4 的缓冲池槽中，如下所示：

第一，以排他模式获取相应描述符的 io_in_progress_lock。

第二，将相应描述符的 IO_IN_PROGRESS 标记位设置为 1，以防其他进程访问。

第三，将所需的页面数据从存储加载到缓冲池插槽中。

第四，更改相应描述符的状态，将 IO_IN_PROGRESS 标记位设置为"0"，且 VALID 标记位设置为"1"。

第五，释放 io_in_progress_lock。

(6) 释放相应分区的 BufMappingLock。

(7) 访问 buffer_id=4 的缓冲池槽。

将页面从存储加载到受害者缓冲池槽

缓冲区管理器将执行以下步骤：

(1) 创建所需页面的 buffer_tag 并查找缓冲表。在本例中假设 buffer_tag 是'Tag_M'(且相应的页面在缓冲区中找不到)。

(2) 使用时钟扫描算法选择一个受害者缓冲池槽位，从缓冲表中获取包含着受害者槽位 buffer_id 的旧表项，并在缓冲区描述符层将受害者槽位的缓冲区描述符钉住。本例中受害者槽的 buffer_id=5，旧表项为 Tag_F,id = 5。时钟扫描将在下一节介绍。

(3) 如果受害者页面是脏页，则将其刷盘（write & fsync），否则进入步骤（4）。

在使用新数据覆盖脏页之前，必须将脏页写入存储中。脏页的刷盘步骤如下：

第一，获取 buffer_id=5 描述符上的共享 content_lock 和独占 io_in_progress_lock。

第二，更改相应描述符的状态：相应 IO_IN_PROCESS 位设置为"1",JUST_DIRTIED 位设置为"0"。

第三，根据具体情况，调用 XLogFlush() 函数将 WAL 缓冲区上的 WAL 数据写入当前 WAL 段文件（WAL 和 XLogFlush 函数将在第 9 章中介绍）。

第四，将受害者页面的数据刷盘至存储中。

第五，更改相应描述符的状态；将 IO_IN_PROCESS 位设置为"0"，将 VALID 位设置为"1"。

第六，释放 io_in_progress_lock 和 content_lock。

(4) 以排他模式获取缓冲区表中旧表项所在分区上的 BufMappingLock。

(5) 获取新表项所在分区上的 BufMappingLock，并将新表项插入缓冲表：

第一，创建新表项：由 buffer_tag='Tag_M'与受害者的 buffer_id 组成的新表项。

第二，以独占模式获取新表项所在分区上的 BufMappingLock。

第三，将新表项插入缓冲区表中。

(6) 从缓冲表中删除旧表项，并释放旧表项所在分区的 BufMappingLock。

(7) 将目标页面数据从存储加载至受害者槽位，然后用 buffer_id=5 更新描述符的标识字段，将脏位设置为 0，并按流程初始化其他标记位。

(8) 释放新表项所在分区上的 BufMappingLock。

(9) 访问 buffer_id=5 对应的缓冲区槽位。

时钟扫描

该算法是 NFU（Not Frequently Used）算法的变体，开销较少，能高效地选出较少使用的页面。

环形缓冲区

在读写大表时，PostgreSQL 会使用环形缓冲区而不是缓冲池。环形缓冲器是一个很小的临时缓冲区域。当满足下列任一条件时，PostgreSQL 将在共享内存中分配一个环形缓冲区。

1. 批量读取。当扫描关系读取数据的大小超过缓冲池的四分之一时，环形缓冲区的大小为 256 KB。

2. 批量写入，当执行下列 SQL 命令时，环形缓冲区大小为 16 MB。

- COPY FROM 命令。
- CREATE TABLE AS 命令。
- CREATE MATERIALIZED VIEW 或 REFRESH MATERIALIZED VIEW 命令。
- ALTER TABLE 命令。

3. 清理过程，当自动清理守护进程执行清理过程时，环形缓冲区大小为 256 KB。

分配的环形缓冲区将在使用后被立即释放。

环形缓冲区的好处显而易见，如果后端进程在不使用环形缓冲区的情况下读取大表，则所有存储在缓冲池中的页面都会被移除，这会导致缓存命中率降低。环形缓冲区可以避免此问题。

为什么批量读取和清理过程的默认环形缓冲区大小为 256 KB

源代码中缓冲区管理器目录下的 README 中解释了这个问题。

顺序扫描使用 256KB 的环形缓冲区，它足够小，因而能放入 L2 缓存中，从而使得操作系统缓存到共享缓冲区的页面传输变得高效。通常更小一点也可以，但环形缓冲区需要足够大到能同时容纳扫描中被钉住的所有页面。

进程与内存架构

进程架构

PostgreSQL 是一个客户端/服务器风格的关系型数据库管理系统，采用多进程架构，运行在单台主机上。

postgres 服务器进程是 PostgreSQL 服务器中所有进程的父进程。

带 start 参数执行 pg_ctl 实用程序会启动一个 postgres 服务器进程。它会在内存中分配共享内存区域，启动各种后台进程。

每当接收到来自客户端的连接请求时，它都会启动一个后端进程，然后由启动的后端进程处理该客户端发出的所有查询。

后端进程

每个后端进程（也称为“postgres”）由 postgres 服务器进程启动，并处理连接另一侧的客户端发出的所有查询。它通过单条 TCP 连接与客户端通信，并在客户端断开连接时终止。

因为一条连接只允许操作一个数据库，所以必须在连接到 PostgreSQL 服务器时显式地指定要连接的数据库。

PostgreSQL 允许多个客户端同时连接，配置参数 max_connections 用于控制最大客户端连接数（默认为 100）。

内存架构

PostgreSQL 的内存架构可以分为两个部分：

本地内存区域——由每个后端进程分配，供自己使用。

共享内存区域——供 PostgreSQL 服务器的所有进程使用。

本地内存区域

每个后端进程都会分配一块本地内存区域用于查询处理。该区域会分为几个子区域——子区域的大小有的固定，有的可变。

共享内存区域

PostgreSQL 服务器启动时会分配共享内存区域，该区域分为几个固定大小的子区域。

PostgreSQL 还分配了以下几个区域：

用于访问控制机制的子区域（例如信号量、轻量级锁、共享和排他锁等）。

各种后台进程使用的子区域，例如 checkpointer 和 autovacuum。

用于事务处理的子区域，例如保存点与两阶段提交（2PC）。