Chapter 8

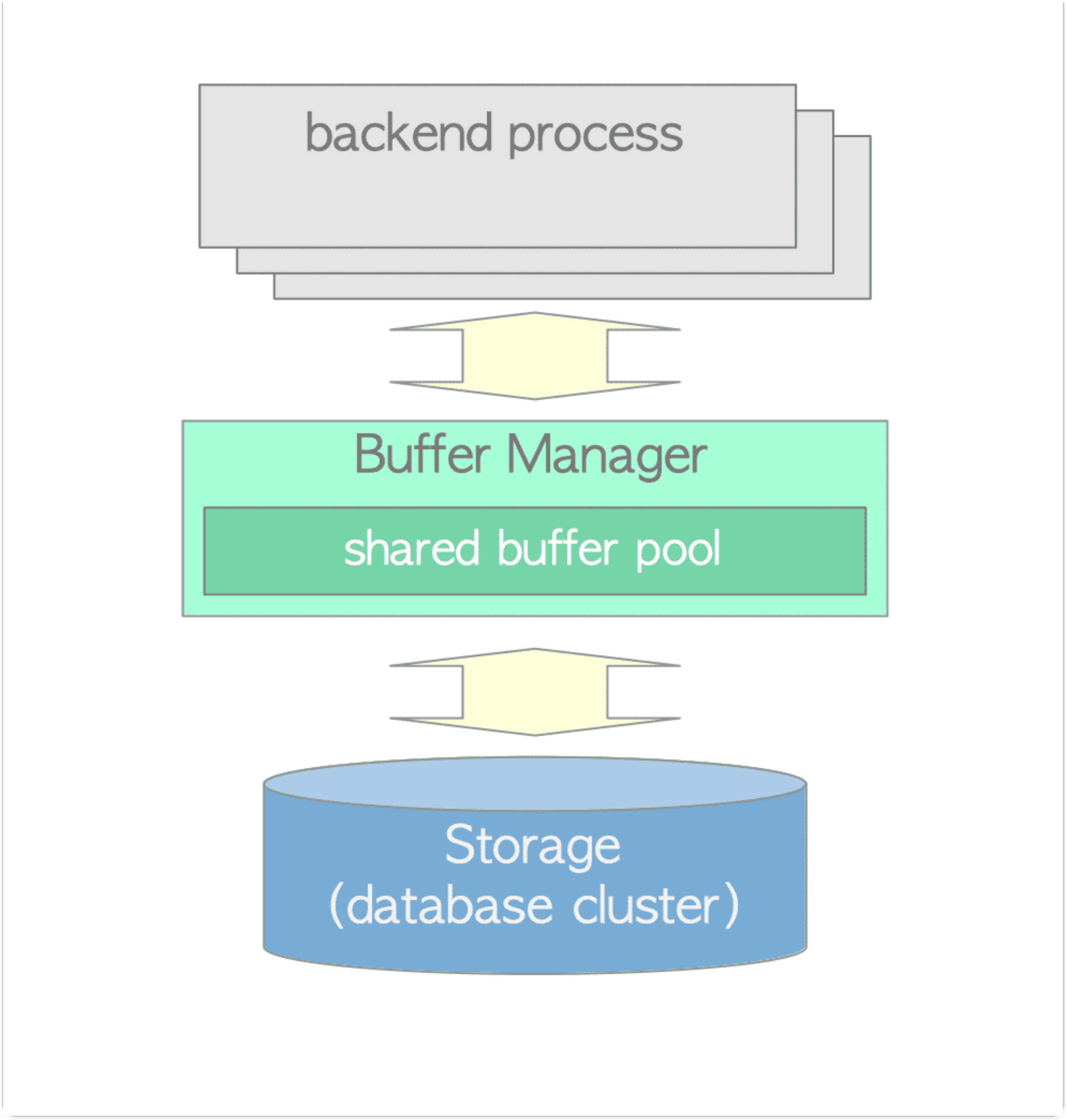
Buffer Manager

The buffer manager manages data transfers between shared memory and persistent storage, and it can have a significant impact on the performance of the DBMS. The PostgreSQL buffer manager works very efficiently.

In this chapter describes the PostgreSQL buffer manager. The first section provides an overview, and the subsequent sections describe the following topics:

- Buffer manager structure
- Buffer manager locks
- How the buffer manager works
- Ring buffer
- Flushing of dirty pages

Fig. 8.1. Relations between buffer manager, storage, and backend processes.



8.1. Overview

This section introduces key concepts that are necessary to understand the descriptions in the subsequent sections.

8.1.1. Buffer Manager Structure

The PostgreSQL buffer manager comprises a buffer table, buffer descriptors, and buffer pool, which are described in the next section. The **buffer pool** layer stores data file pages, such as tables and indexes, as well as freespace maps and visibility maps. The buffer pool is an array, where each slot stores one page of a data file. The Indices of a buffer pool array are referred to as **buffer_id**s.

Sections 8.2 and 8.3 describe the details of the buffer manager internals.

8.1.2. Buffer Tag

In PostgreSQL, each page of all data files can be assigned a unique tag, i.e. a **buffer tag**. When the buffer manager receives a request, PostgreSQL uses the buffer_tag of the desired page.

The buffer_tag has five values:

- **specOid**: The OID of the tablespace to which the relation containing the target page belongs.
- **dbOid**: The OID of the database to which the relation containing the target page belongs.
- **relNumber**: The number of the relation file that contains the target page.
- **blockNum**: The block number of the target page in the relation.
- **forkNum**: The fork number of the relation that the page belongs to. The fork numbers of tables, freespace maps, and visibility maps are defined in 0, 1 and 2, respectively.

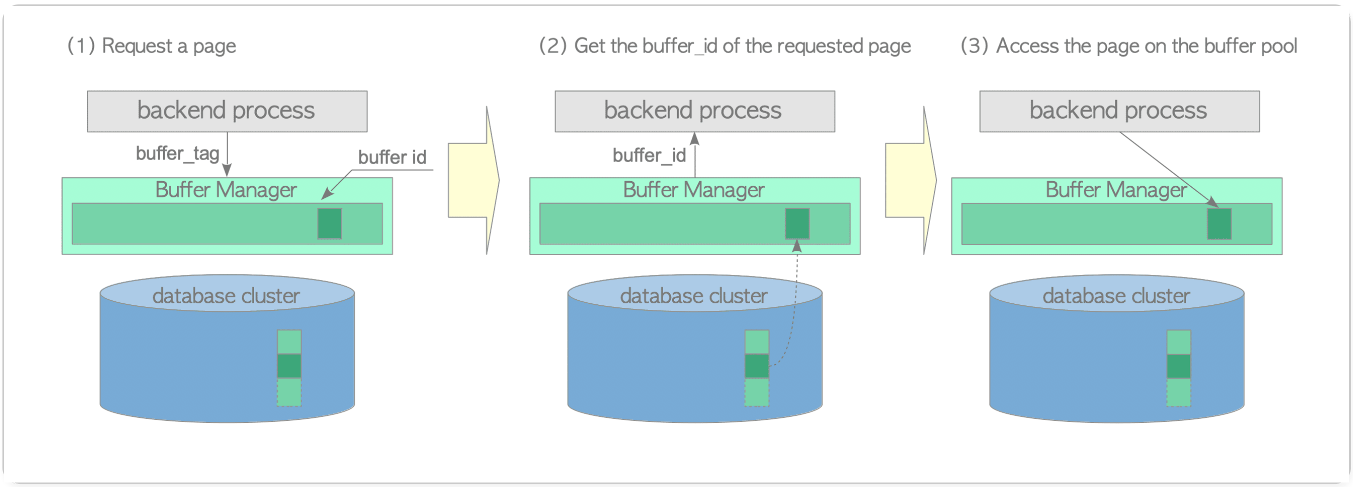
For example, the buffer_tag '{16821, 16384, 37721, 0, 7}' identifies the page that is in the seventh block of the table whose OID and fork number are 37721 and 0, respectively. The table is contained in the database whose OID is 16384 under the tablespace whose OID is 16821.

Similarly, the buffer_tag '{16821, 16384, 37721, 1, 3}' identifies the page that is in the third block of the freespace map whose OID and fork number are 37721 and 1, respectively.

8.1.3. How a Backend Process Reads Pages

This subsection describes how a backend process reads a page from the buffer manager (Fig. 8.2).

Fig. 8.2. How a backend reads a page from the buffer manager.



(1) When reading a table or index page, a backend process sends a request that includes the page's buffer_tag to the buffer manager.
(2) The buffer manager returns the buffer_ID of the slot that stores the requested page. If the requested page is not stored in the buffer pool, the buffer manager loads the page from persistent storage to one of the buffer pool slots and then returns the buffer_ID of the slot.
(3) The backend process accesses the buffer_ID's slot (to read the desired page).

When a backend process modifies a page in the buffer pool (e.g., by inserting tuples), the modified page, which has not yet been flushed to storage, is referred to as a **dirty page**.

Section 8.4 describes how the buffer manager works in mode detail.

8.1.4. Page Replacement Algorithm

When all buffer pool slots are occupied and the requested page is not stored, the buffer manager must select one page in the buffer pool to be replaced by the requested page. Typically, in the field of computer science, page selection algorithms are called *page replacement algorithms*, and the selected page is referred to as a **victim page**.

Research on page replacement algorithms has been ongoing since the advent of computer science. Many replacement algorithms have been proposed, and PostgreSQL has used the **clock sweep** algorithm since version 8.1. Clock sweep is simpler and more efficient than the LRU algorithm used in previous versions.

Section 8.4.4 describes the details of clock sweep.

8.1.5. Flushing Dirty Pages

Dirty pages should eventually be flushed to storage. However, the buffer manager requires help to perform this task. In PostgreSQL, two background processes, **checkpointer** and **background writer**, are responsible for this task.

Section 8.6 describes the checkpointer and background writer.

Direct I/O

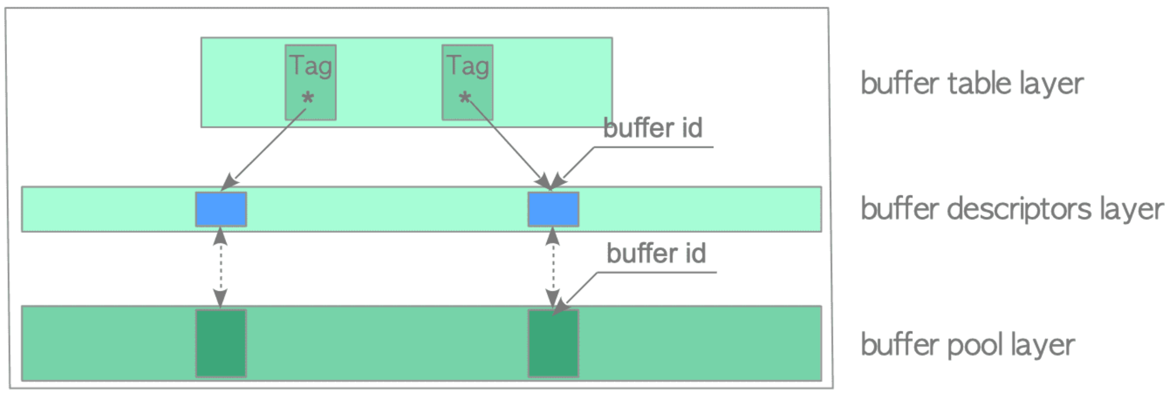
PostgreSQL versions 15 and earlier do not support direct I/O, although it has been discussed. Reffer to this discussion on the pgsql-ML and this article.

In version 16, the debug-io-direct option has been added. This option is for developers to improve the use of direct I/O in PostgreSQL. If development goes well, direct I/O will be officially supported in the near future.

8.2. Buffer Manager Structure

The PostgreSQL buffer manager comprises three layers: the *buffer table*, *buffer descriptors*, and *buffer pool* (Fig. 8.3):

Fig. 8.3. Buffer manager's three-layer structure.



- **Buffer pool**: An array that stores data file pages. Each slot in the array is referred to as *buffer_id*s.

- **Buffer descriptors**: An array of buffer descriptors. Each descriptor has a one-to-one correspondence to a buffer pool slot and holds the metadata of the stored page in the corresponding slot.
  Note that the term 'buffer descriptors layer' has been adopted for convenience and is only used in this document.
- **Buffer table**: A hash table that stores the relations between the *buffer_tag*s of stored pages and the *buffer_id*s of the descriptors that hold the stored pages' respective metadata.

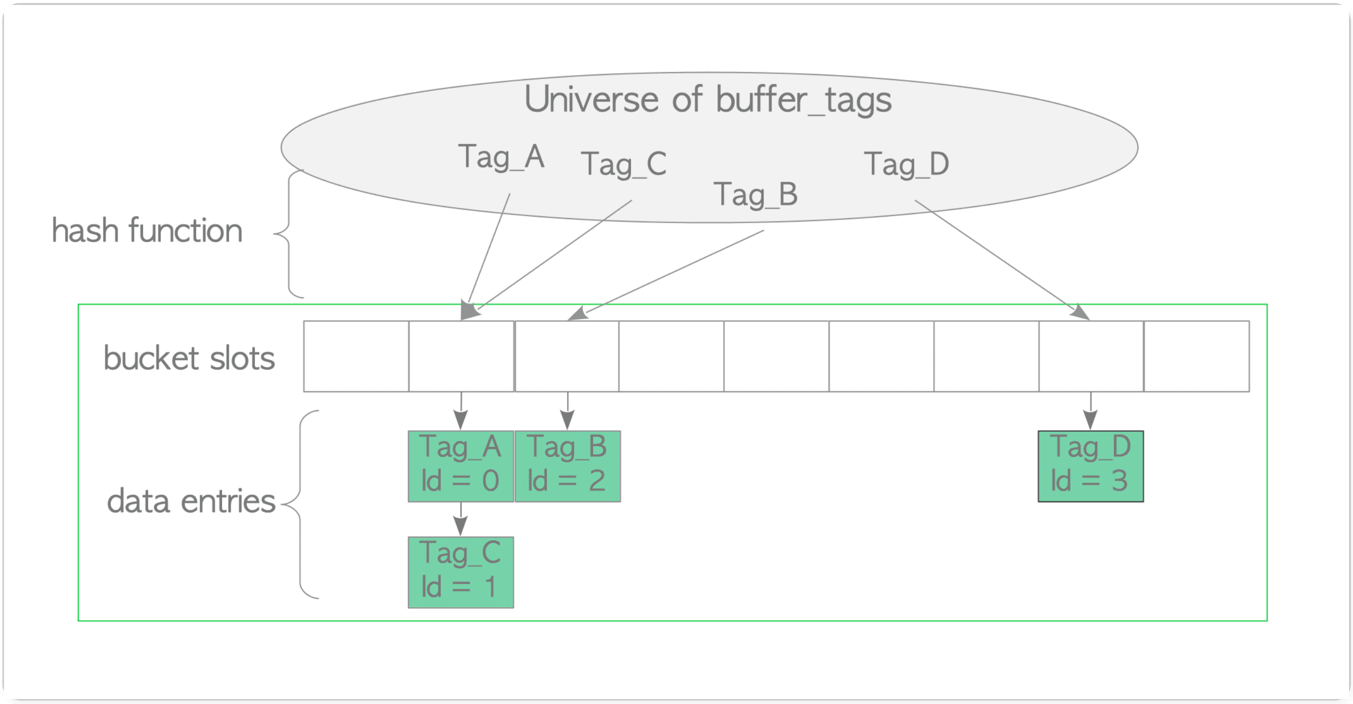
These layers are described in detail in the following subsections.

8.2.1. Buffer Table

A buffer table can be logically divided into three parts: a hash function, hash bucket slots, and data entries (Fig. 8.4).

The built-in hash function maps buffer_tags to the hash bucket slots. Even though the number of hash bucket slots is greater than the number of buffer pool slots, collisions may occur. Therefore, the buffer table uses a *separate chaining with linked lists* method to resolve collisions. When data entries are mapped to the same bucket slot, this method stores the entries in the same linked list, as shown in Fig. 8.4.

Fig. 8.4. Buffer table.



A data entry comprises two values: the buffer_tag of a page, and the buffer_id of the descriptor that holds the page's metadata. For example, a data entry '*Tag_A, id=1*' means that the buffer descriptor with buffer_id *1* stores metadata of the page tagged with *Tag_A*.

Hash function

The hash function is a composite function of calc_bucket() and hash(). The following is its representation as a pseudo-function.

```
uint32 bucket_slot = calc_bucket(unsigned hash(BufferTag buffer_tag), uint32 bucket_size)
```

Note: Basic operations (lookup, insertion, and deletion of data entries) are not explained here. These are very common operations and are explained in the following sections.

8.2.2. Buffer Descriptor

The structure of buffer descriptors has been improved in version 9.6. First, I explain buffer descriptors from versions 9.5 and earlier, and then I explain how buffer descriptors from version 9.6 and later differ from previous versions.

The buffer descriptors layer is described in the next subsection.

8.2.2.1. Versions 9.5 or earlier

The buffer descriptor structure in versions 9.5 and earlier holds the metadata of the stored page in the corresponding buffer pool slot. The buffer descriptor structure is defined by the BufferDesc structure. The following are some of the main fields:

```
/*
 * Flags for buffer descriptors
 *
 * Note: TAG_VALID essentially means that there is a buffer hashtable
 * entry associated with the buffer's tag.
 */
#define BM_DIRTY                (1 << 0)    /* data needs writing */
#define BM_VALID                (1 << 1)    /* data is valid */
#define BM_TAG_VALID            (1 << 2)    /* tag is assigned */
#define BM_IO_IN_PROGRESS       (1 << 3)    /* read or write in progress */
#define BM_IO_ERROR             (1 << 4)    /* previous I/O failed */
#define BM_JUST_DIRTIED         (1 << 5)    /* dirtied since write started */
#define BM_PIN_COUNT_WAITER     (1 << 6)    /* have waiter for sole pin */
#define BM_CHECKPOINT_NEEDED    (1 << 7)    /* must write for checkpoint */
#define BM_PERMANENT            (1 << 8)    /* permanent relation (not unlogged) */

src/include/storage/buf_internals.h
typedef struct sbufdesc
{
   BufferTag    tag;                 /* ID of page contained in buffer */
   BufFlags     flags;               /* see bit definitions above */
   uint16       usage_count;         /* usage counter for clock sweep code */
   unsigned     refcount;            /* # of backends holding pins on buffer */
   int          wait_backend_pid;    /* backend PID of pin-count waiter */
   slock_t      buf_hdr_lock;        /* protects the above fields */
   int          buf_id;              /* buffer's index number (from 0) */
   int          freeNext;            /* link in freelist chain */

   LWLockId     io_in_progress_lock; /* to wait for I/O to complete */
   LWLockId     content_lock;        /* to lock access to buffer contents */
} BufferDesc;
```

- **tag** holds the *buffer_tag* of the stored page in the corresponding buffer pool slot. (buffer tag is defined in Section 8.1.2.)
- **buf_id** identifies the descriptor. It is equivalent to the *buffer_id* of the corresponding buffer pool slot.
- **refcount** holds the number of PostgreSQL processes that are currently accessing the associated stored page. It is also referred to as **pin count**.

When a PostgreSQL process accesses the stored page, the refcount must be incremented by 1 (refcount++). After accessing the page, the refcount must be decreased by 1 (refcount--). When the refcount is zero, the associated stored page is **unpinned**, meaning it is not currently being accessed. Otherwise, it is **pinned**.

- **usage_count** holds the number of times the associated stored page has been accessed since it was loaded into the corresponding buffer pool slot. It is used in the page replacement algorithm (Section 8.4.4).
- **content_lock** and **io_in_progress_lock** are light-weight locks that are used to control access to the associated stored page. These fields are described in Section 8.3.2.
- **flags** can hold several states of the associated stored page. The main states are as follows:
  - **dirty bit** indicates that the stored page is dirty.
  - **valid bit** indicates whether the stored page is valid, meaning it can be read or written.
    If this bit is *valid*, then the corresponding buffer pool slot stores a page and the descriptor holds the page metadata, and the stored page can be read or written.
    If this bit is *invalid*, then the descriptor does not hold any metadata and the stored page cannot be read or written.
  - **io_in_progress bit** indicates whether the buffer manager is reading or writing the associated page from or to storage.
- **buf_hdr_lock** is a spin lock that protects the fields: flags, usage_count, refcount.
- **freeNext** is a pointer to the next descriptor to generate a *freelist*, which is described in the next subsection.

8.2.2.2. Versions 9.6 or later

The buffer descriptor structure is defined by the BufferDesc structure.

```
/*
 * Flags for buffer descriptors
 *
 * Note: BM_TAG_VALID essentially means that there is a buffer hashtable
 * entry associated with the buffer's tag.
 */
#define BM_LOCKED               (1U << 22)      /* buffer header is locked */
#define BM_DIRTY                (1U << 23)      /* data needs writing */
#define BM_VALID                (1U << 24)      /* data is valid */
#define BM_TAG_VALID            (1U << 25)      /* tag is assigned */
#define BM_IO_IN_PROGRESS       (1U << 26)      /* read or write in progress */
#define BM_IO_ERROR             (1U << 27)      /* previous I/O failed */
#define BM_JUST_DIRTIED         (1U << 28)      /* dirtied since write started */
#define BM_PIN_COUNT_WAITER     (1U << 29)      /* have waiter for sole pin */
#define BM_CHECKPOINT_NEEDED    (1U << 30)      /* must write for checkpoint */
#define BM_PERMANENT            (1U << 31)      /* permanent buffer (not unlogged,
                                                 * or init fork) */

#define PG_HAVE_ATOMIC_U32_SUPPORT
typedef struct pg_atomic_uint32
{
        volatile uint32 value;
} pg_atomic_uint32;


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_pgprocno;  /* backend of pin-count waiter */
        int             freeNext;               /* link in freelist chain */
        LWLock          content_lock;   /* to lock access to buffer contents */
 } BufferDesc;
```

- **tag** holds the *buffer_tag* of the stored page in the corresponding buffer pool slot.
- **buf_id** identifies the descriptor.
- **content_lock** is a light-weight lock that is used to control access to the associated stored page.
- **freeNext** is a pointer to the next descriptor to generate a *freelist*.
- **states** can hold several states and variables of the associated stored page, such as refcount and usage_count.

The flags, usage_count, and refcount fields have been combined into a single 32-bit data (states) to use the CPU atomic operations. Therefore, the *io_in_progress_lock* and spin lock (buf_hdr_lock) have been removed since there is no longer a need to protect these values.

Atomic Operations

Instead of exclusive control of data by software, CPU atomic operations use hardware-enforced exclusive control and atomic read-modify-write operations to perform efficiently.

The structure *BufferDesc* is defined in src/include/storage/buf_internals.h.

8.2.2.3. Descriptor States

To simplify the following descriptions, three descriptor states are defined:

Empty: When the corresponding buffer pool slot does not store a page (i.e. *refcount* and *usage_count* are 0), the state of this descriptor is *empty*.

Pinned: When the corresponding buffer pool slot stores a page and any PostgreSQL processes are accessing the page (i.e. *refcount* and *usage_count* are greater than or equal to 1), the state of this buffer descriptor is *pinned*.

Unpinned: When the corresponding buffer pool slot stores a page but no PostgreSQL processes are accessing the page (i.e. *usage_count* is greater than or equal to 1, but *refcount* is 0), the state of this buffer descriptor is *unpinned*.

Each descriptor will have one of the above states. The descriptor state changes depending on certain conditions, which are described in the next subsection.

In the following figures, buffer descriptors' states are represented by coloured boxes.

☐ (white) *Empty*

☐ (blue) *Pinned*
☐ (aqua blue) *Unpinned*

In addition, a dirty page is denoted as 'X'. For example, an unpinned dirty descriptor is represented by $\boxed{X}$.

8.2.3. Buffer Descriptors Layer

A collection of buffer descriptors forms an array, which is referred to as the *buffer descriptors layer* in this document.

When the PostgreSQL server starts, the state of all buffer descriptors is *empty*. In PostgreSQL, those descriptors comprise a linked list called **freelist** (Fig. 8.5).

It is important to note that the **freelist** in PostgreSQL is completely different concept from the *freelists* in Oracle. The freelist in PostgreSQL is simply linked list of empty buffer descriptors. In PostgreSQL, *freespace maps*, which are described in Section 5.3.4, serve as the same purpose as the freelists in Oracle.

Fig. 8.5. Buffer manager initial state.

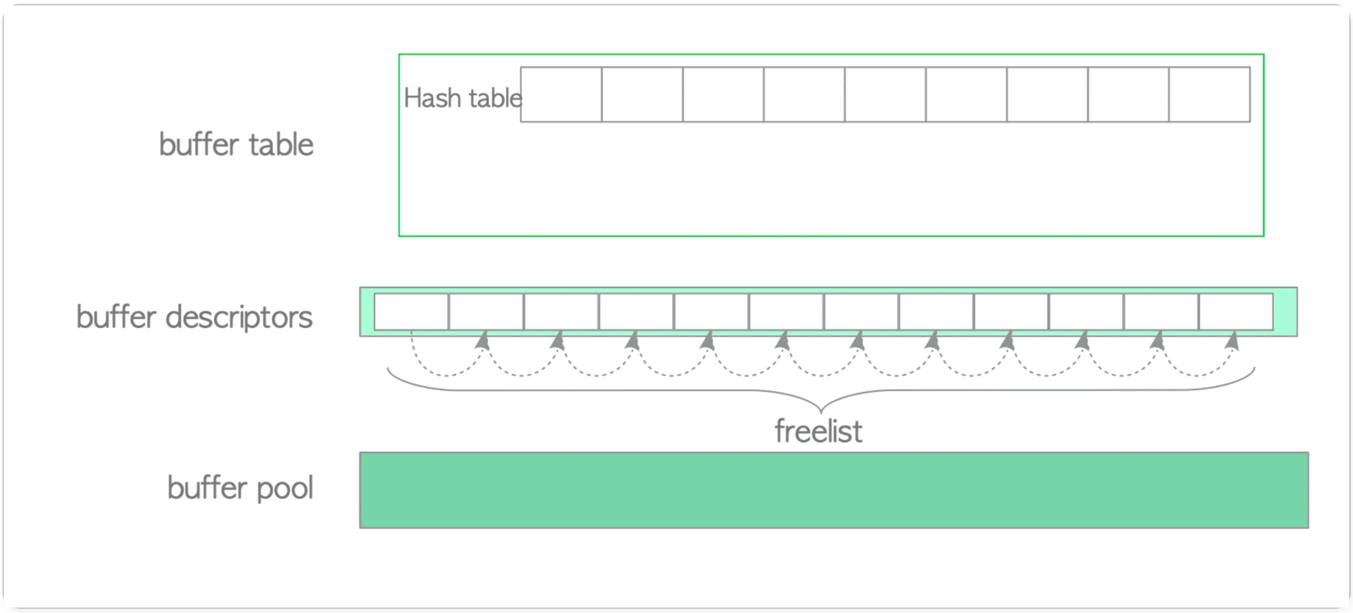
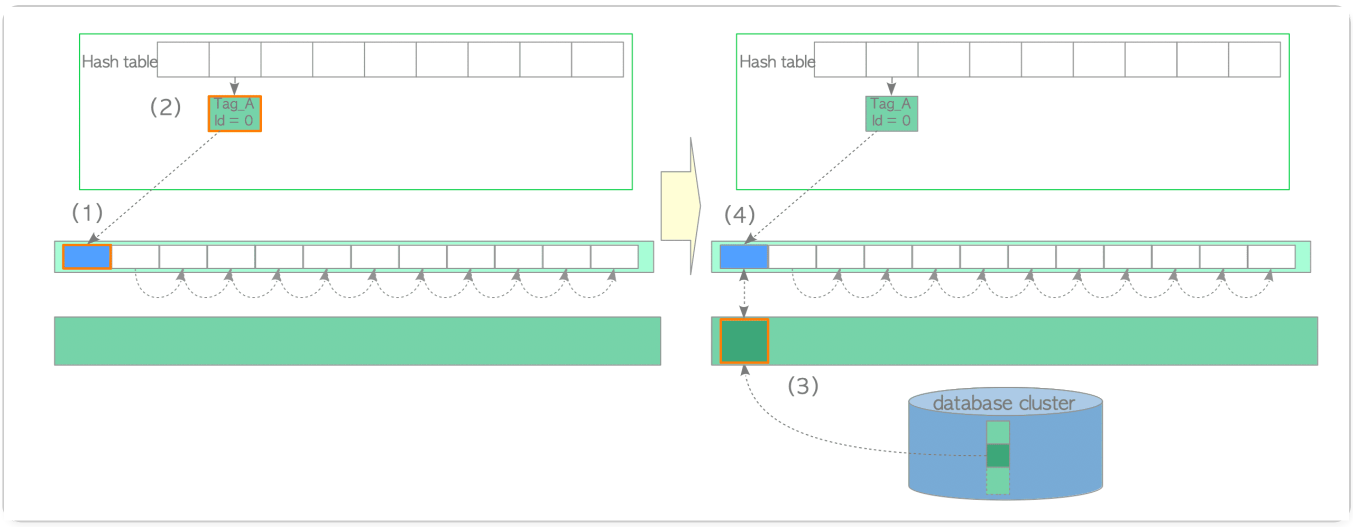


Figure 8.6 shows that how the first page is loaded.

(1) Retrieve an empty descriptor from the top of the freelist, and pin it (i.e. increase its refcount and usage_count by 1).
(2) Insert a new entry into the buffer table that maps the tag of the first page to the buffer_id of the retrieved descriptor.
(3) Load the new page from storage into the corresponding buffer pool slot.
(4) Save the metadata of the new page to the retrieved descriptor.

The second and subsequent pages are loaded in a similar manner. Additional details are provided in Section 8.4.2.

Fig. 8.6. Loading the first page.



Descriptors that have been retrieved from the freelist always hold page's metadata. In other words, non-empty descriptors do not return to the freelist once they have been used. However, the corresponding descriptors are added to the freelist again and the descriptor state is set to 'empty' when one of the following occurs:

1. Tables or indexes are dropped.
2. Databases are dropped.
3. Tables or indexes are cleaned up using the VACUUM FULL command.

Why empty descriptors comprise the freelist?

The freelist is created to allow for the immediate retrieval of the first descriptor. This is a usual practice for dynamic memory resource allocation. For more information, please refer to this description.

The buffer descriptors layer contains an unsigned 32-bit integer variable, i.e. **nextVictimBuffer**. This variable is used in the page replacement algorithm described in Section 8.4.4.

8.2.4. Buffer Pool

The buffer pool is a simple array that stores data file pages, such as tables and indexes. The indices of the buffer pool array are called *buffer_id*s.

The buffer pool slot size is 8 KB, which is equal to the size of a page. Therefore, each slot can store an entire page.

8.3. Buffer Manager Locks

The buffer manager uses many locks for a variety of purposes. This section describes the locks that are necessary for the explanations in the subsequent sections.

Note that the locks described in this section are part of a synchronization mechanism for the buffer manager. They do **not** relate to any SQL statements or SQL options.

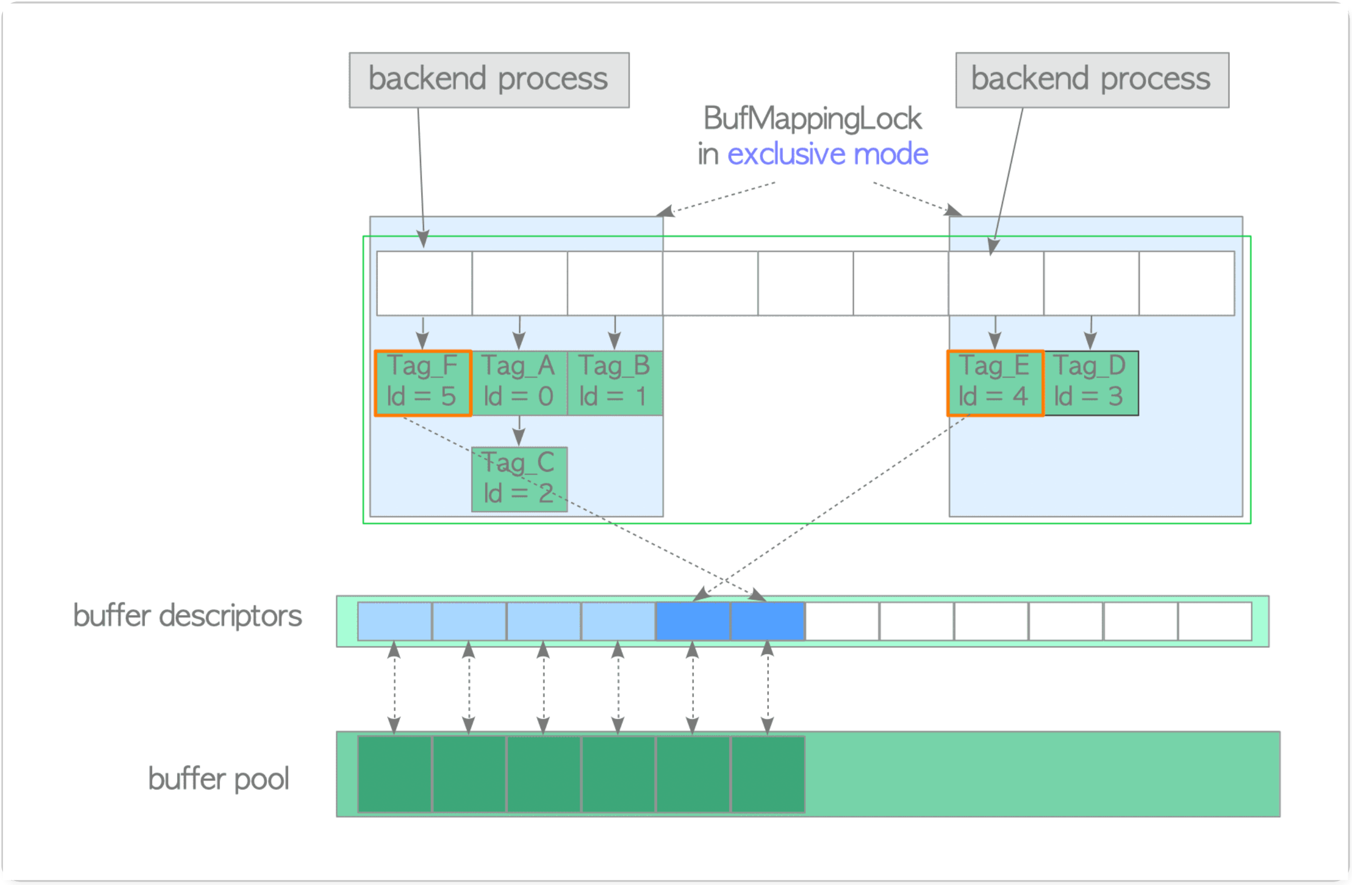
8.3.1. Buffer Table Locks

BufMappingLock protects the data integrity of the entire buffer table. It is a light-weight lock that can be used in both shared and exclusive modes. When searching an entry in the buffer table, a backend process holds a shared BufMappingLock. When inserting or deleting entries, a backend process holds an exclusive lock.

The BufMappingLock is split into partitions to reduce contention in the buffer table (the default is 128 partitions). Each BufMappingLock partition guards a portion of the corresponding hash bucket slots.

Figure 8.7 shows a typical example of the effect of splitting BufMappingLock. Two backend processes can simultaneously hold respective BufMappingLock partitions in exclusive mode to insert new data entries. If BufMappingLock were a single system-wide lock, both processes would have to wait for the other process to finish, depending on which process started first.

Fig. 8.7. Two processes simultaneously acquire the respective partitions of BufMappingLock in exclusive mode to insert new data entries.



The buffer table requires many other locks. For example, the buffer table internally uses a spin lock to delete an entry. However, descriptions of these other locks are omitted because they are not required in this document.

Until PostgreSQL version 9.4, BufMappingLock was split into 16 separate locks by default.

8.3.2. Locks for Each Buffer Descriptor

In versions 9.5 or earlier, each buffer descriptor used two lightweight locks, **content_lock** and **io_in_progress_lock**, to control access to the stored page in the corresponding buffer pool slot. A spinlock (buf_hdr_lock) was used when the values of its own fields (i.e., usage_count, refcount, flags) were checked or changed.

In version 9.6, buffer access methods have been improved. The io_in_progress_lock and spin lock (buf_hdr_lock) have been removed. Instead of using these locks, versions 9.6 and later use CPU atomic operations to inspect and change their values.

8.3.2.1. content_lock

The content_lock is a typical lock that enforces access restrictions. It can be used in *shared* and *exclusive* modes.

When reading a page, a backend process acquires a shared content_lock of the buffer descriptor that stores the page.

An exclusive content_lock is acquired when doing one of the following:

- Inserting rows (i.e., tuples) into the stored page or changing the t_xmin/t_xmax fields of tuples within the stored page. (t_xmin and t_xmax are described in Section 5.2; simply, when deleting or updating rows, these fields of the associated tuples are changed).
- Physically removing tuples or compacting free space on the stored page. (This is performed by vacuum processing and HOT, which are described in Chapters 6 and 7, respectively).
- Freezing tuples within the stored page. (Freezing is described in Section 5.10.1 and Section 6.3).

The official README file provides more details.

8.3.2.2. io_in_progress_lock (versions 9.5 or earlier)

In versions 9.5 or earlier, the io_in_progress lock was used to wait for I/O on a buffer to complete. When a PostgreSQL process loads or writes page data from or to storage, the process acquires an exclusive io_in_progress lock of the corresponding descriptor while accessing the storage.

8.3.2.3. spinlock (versions 9.5 or earlier)

When the flags or other fields (such as refcount and usage_count) are checked or changed, a spinlock was used. Two specific examples of spinlock usage are given below:

(1) Pinning a buffer descriptor:
1. Acquire a spinlock of the buffer descriptor.
2. Increase the values of its refcount and usage_count by 1.
3. Release the spinlock.

```
LockBufHdr(bufferdesc);    /* Acquire a spinlock */
bufferdesc->refcont++;
bufferdesc->usage_count++;
UnlockBufHdr(bufferdesc); /* Release the spinlock */
```

(2) Setting the dirty bit to '1':

1. Acquire a spinlock of the buffer descriptor.
2. Set the dirty bit to '1' using a bitwise operation.
3. Release the spinlock.

```
#define BM_DIRTY             (1 << 0)    /* data needs writing */
#define BM_VALID             (1 << 1)    /* data is valid */
#define BM_TAG_VALID         (1 << 2)    /* tag is assigned */
#define BM_IO_IN_PROGRESS    (1 << 3)    /* read or write in progress */
#define BM_JUST_DIRTIED      (1 << 5)    /* dirtied since write started */

LockBufHdr(bufferdesc);
bufferdesc->flags |= BM_DIRTY;
UnlockBufHdr(bufferdesc);
```

Changing other bits is performed in the same manner.

8.4. How the Buffer Manager Works

This section describes how the buffer manager works. When a backend process wants to access a desired page, it calls the *ReadBufferExtended* function.

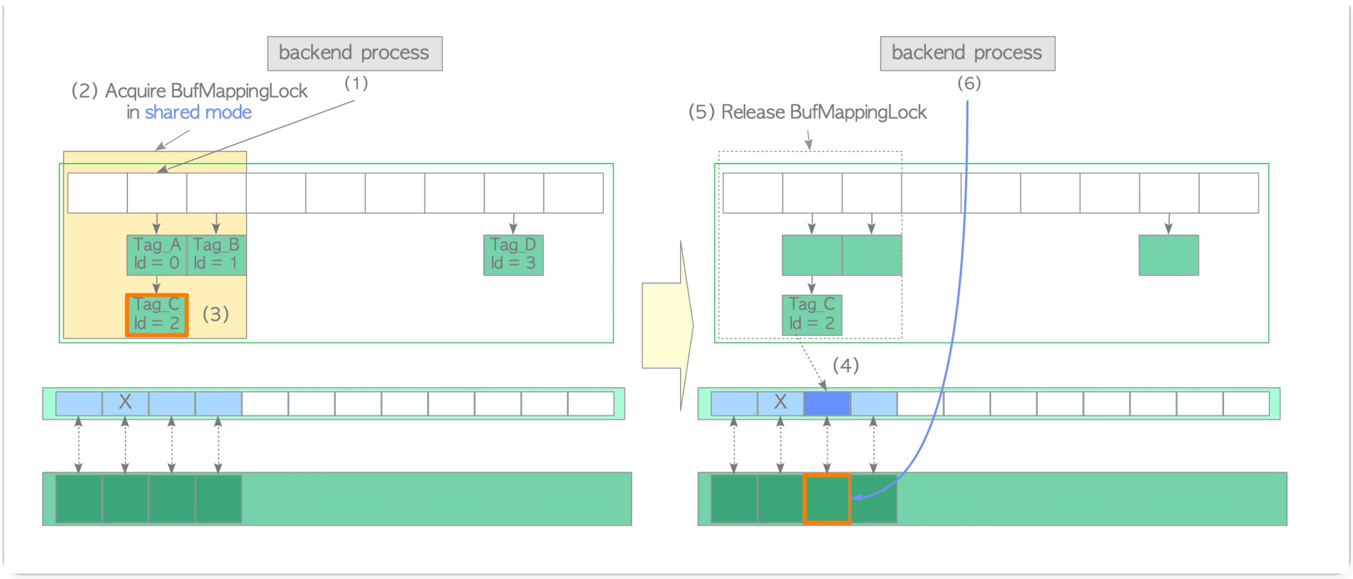
The behavior of the *ReadBufferExtended* function depends on three logical cases. Each case is described in the following subsections. In addition, the PostgreSQL *clock sweep* page replacement algorithm is described in the final subsection.

8.4.1. Accessing a Page Stored in the Buffer Pool

First, the simplest case is described, in which the desired page is already stored in the buffer pool. In this case, the buffer manager performs the following steps:

(1) Create the *buffer_tag* of the desired page (in this example, the buffer_tag is 'Tag_C') and compute the *hash bucket slot* that contains the associated entry of the created *buffer_tag*, using the hash function.
(2) Acquire the BufMappingLock partition that covers the obtained hash bucket slot in shared mode (this lock will be released in step (5)).
(3) Look up the entry whose tag is 'Tag_C' and obtain the *buffer_id* from the entry. In this example, the buffer_id is 2.
(4) Pin the buffer descriptor for buffer_id 2, increasing the refcount and usage_count of the descriptor by 1. (Section 8.3.2 describes pinning).
(5) Release the BufMappingLock.
(6) Access the buffer pool slot with buffer_id 2.

Fig. 8.8. Accessing a page stored in the buffer pool.



Then, when reading rows from the page in the buffer pool slot, the PostgreSQL process acquires the *shared content_lock* of the corresponding buffer descriptor. Therefore, buffer pool slots can be read by multiple processes simultaneously.

When inserting (and updating or deleting) rows to the page, a Postgres process acquires the *exclusive content_lock* of the corresponding buffer descriptor. (Note that the dirty bit of the page must be set to '1'.)

After accessing the pages, the refcount values of the corresponding buffer descriptors are decreased by 1.

8.4.2. Loading a Page from Storage to Empty Slot

In this second case, assume that the desired page is not in the buffer pool and the freelist has free elements (empty descriptors). In this case, the buffer manager performs the following steps:

(1) Look up the buffer table (we assume that it is not found).
   1. Create the buffer_tag of the desired page (in this example, the buffer_tag is 'Tag_E') and compute the hash bucket slot.
   2. Acquire the BufMappingLock partition in shared mode.
   3. Look up the buffer table. (Not found according to the assumption.)
   4. Release the BufMappingLock.

(2) Obtain the *empty buffer descriptor* from the freelist, and pin it. In this example, the buffer_id of the obtained descriptor is 4.

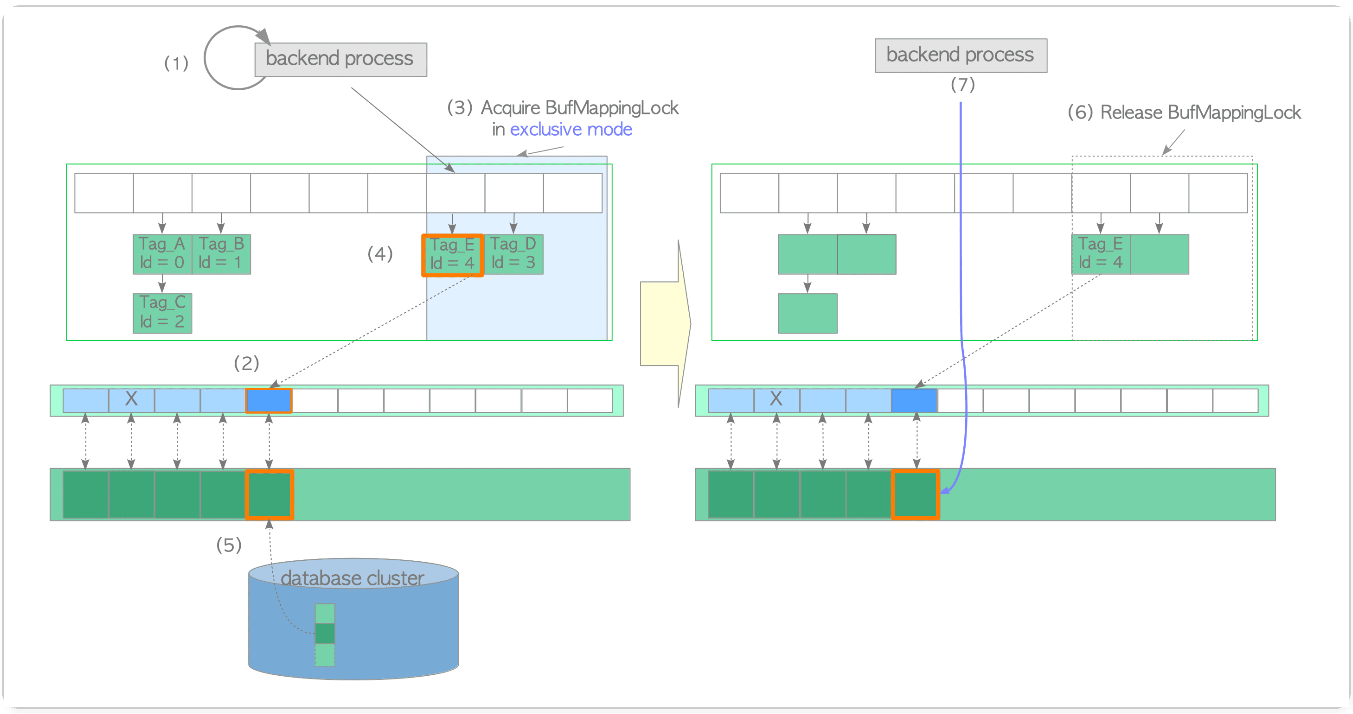
(3) Acquire the BufMappingLock partition in *exclusive* mode. (This lock will be released in step (6).)

(4) Create a new data entry that comprises the buffer_tag 'Tag_E' and buffer_id 4. Insert the created entry to the buffer table.

(5) Load the desired page data from storage to the buffer pool slot with buffer_id 4 as follows:
   1. In versions 9.5 or earlier, acquire the exclusive *io_in_progress_lock* of the corresponding descriptor.
   2. Set the *io_in_progress* bit of the corresponding descriptor to '1' to prevent access by other processes.
   3. Load the desired page data from storage to the buffer pool slot.
   4. Change the states of the corresponding descriptor: the *io_in_progress* bit is set to '0', and the *valid* bit is set to '1'.
   5. In versions 9.5 or earlier, release the *io_in_progress_lock*.

(6) Release the BufMappingLock.
(7) Access the buffer pool slot with buffer_id 4.

Fig. 8.9. Loading a page from storage to an empty slot.



8.4.3. Loading a Page from Storage to a Victim Buffer Pool Slot

In this case, assume that all buffer pool slots are occupied by pages but the desired page is not stored. The buffer manager performs the following steps:

(1) Create the buffer_tag of the desired page and look up the buffer table. In this example, we assume that the buffer_tag is 'Tag_M' (the desired page is not found).

(2) Select a victim buffer pool slot using the clock-sweep algorithm. Obtain the old entry, which contains the buffer_id of the victim pool slot, from the buffer table and pin the victim pool slot in the buffer descriptors layer. In this example, the buffer_id of the victim slot is 5 and the old entry is 'Tag_F, id=5'. The clock sweep is described in the next subsection.

(3) Flush (write and fsync) the victim page data if it is dirty; otherwise proceed to step (4). The dirty page must be written to storage before overwriting with new data. Flushing a dirty page is performed as follows:

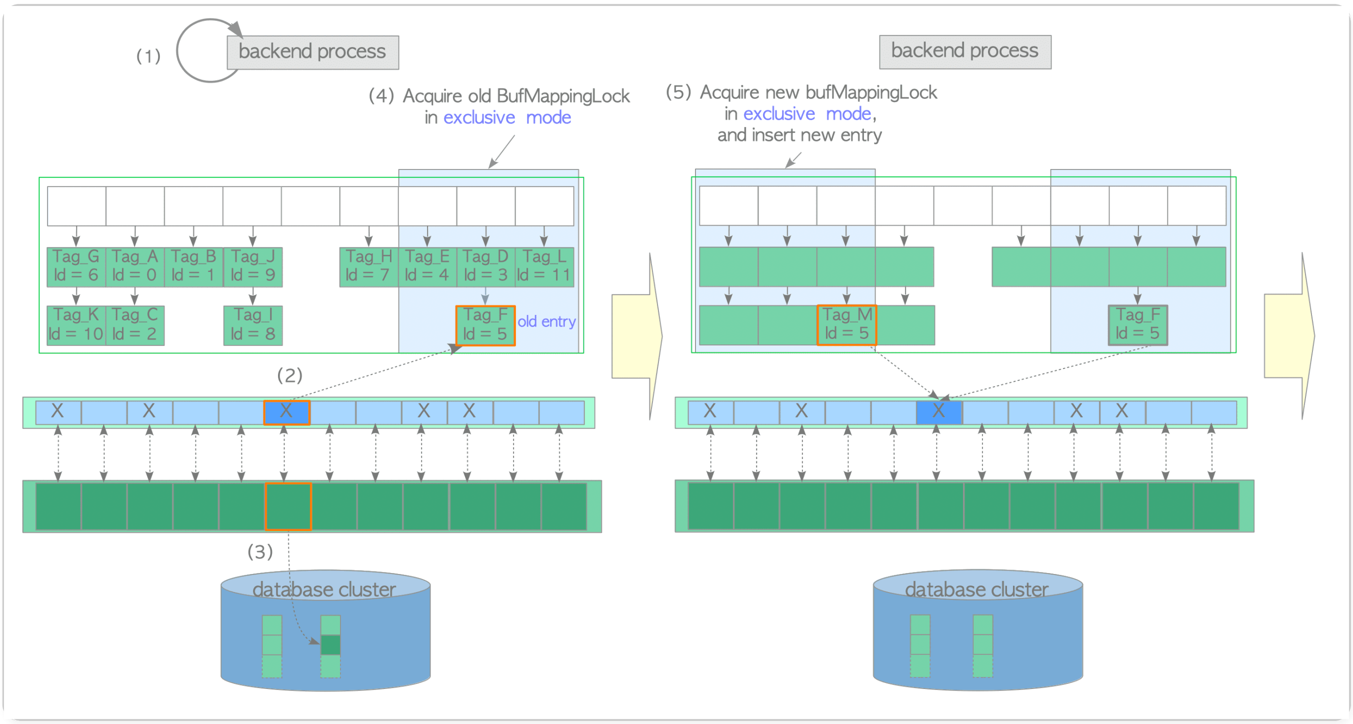
1. Acquire the shared content_lock and the exclusive io_in_progress lock of the descriptor with buffer_id 5 (released in step 6).
2. Change the states of the corresponding descriptor; the *io_in_progress* bit is set to '1' and the *just_dirtied* bit is set to '0'.
3. Depending on the situation, the *XLogFlush()* function is invoked to write WAL data on the WAL buffer to the current WAL segment file (details are omitted; WAL and the *XLogFlush* function are described in Chapter 9).
4. Flush the victim page data to storage.
5. Change the states of the corresponding descriptor; the *io_in_progress* bit is set to '0' and the *valid* bit is set to '1'.
6. Release the io_in_progress and content_lock locks.

(4) Acquire the old BufMappingLock partition that covers the slot that contains the old entry, in exclusive mode.

(5) Acquire the new BufMappingLock partition and insert the new entry to the buffer table:

1. Create the new entry comprised of the new buffer_tag 'Tag_M' and the victim's buffer_id.
2. Acquire the new BufMappingLock partition that covers the slot containing the new entry in exclusive mode.
3. Insert the new entry to the buffer table.

Fig. 8.10. Loading a page from storage to a victim buffer pool slot.



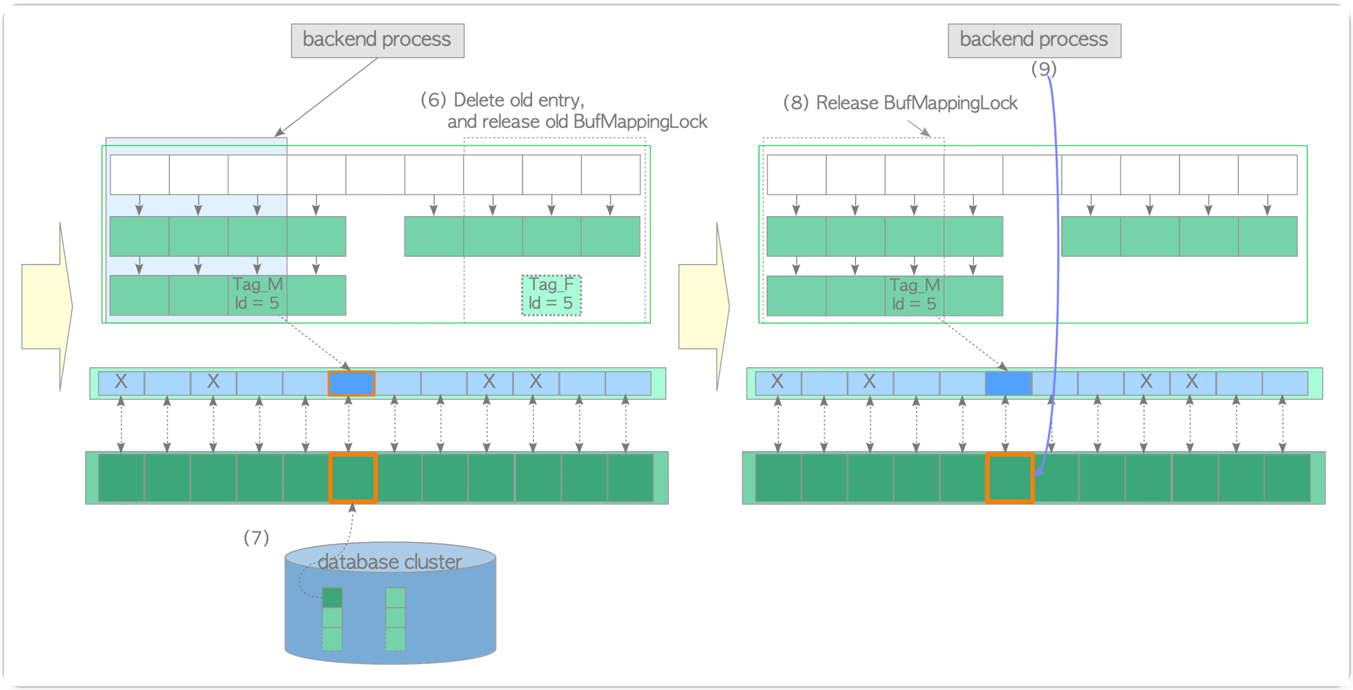
(6) Delete the old entry from the buffer table, and release the old BufMappingLock partition.

(7) Load the desired page data from the storage to the victim buffer slot. Then, update the flags of the descriptor with buffer_id 5; the dirty bit is set to 0 and other bits are initialized.

(8) Release the new BufMappingLock partition.

(9) Access the buffer pool slot with buffer_id 5.

Fig. 8.11. Loading a page from storage to a victim buffer pool slot (continued from Fig. 8.10).



8.4.4. Page Replacement Algorithm: Clock Sweep

The rest of this section describes the **clock-sweep** algorithm. This algorithm is a variant of NFU (Not Frequently Used) with low overhead; it selects less frequently used pages efficiently.

Imagine buffer descriptors as a circular list (Fig. 8.12). The nextVictimBuffer, an unsigned 32-bit integer, is always pointing to one of the buffer descriptors and rotates clockwise. The pseudocode and description of the algorithm are follows:

</> Pseudocode: clock-sweep

```
     WHILE true
(1)     Obtain the candidate buffer descriptor pointed by the nextVictimBuffer
(2)     IF the candidate descriptor is unpinned THEN
(3)         IF the candidate descriptor's usage_count == 0 THEN
                BREAK WHILE LOOP  /* the corresponding slot of this descriptor is victim slot. */
            ELSE
                Decrease the candidate descriptpor's usage_count by 1
            END IF
        END IF
(4)     Advance nextVictimBuffer to the next one
      END WHILE
(5) RETURN buffer_id of the victim
```

(1) Obtain the candidate buffer descriptor pointed to by *nextVictimBuffer*.

(2) If the candidate buffer descriptor is *unpinned*, proceed to step (3). Otherwise, proceed to step (4).

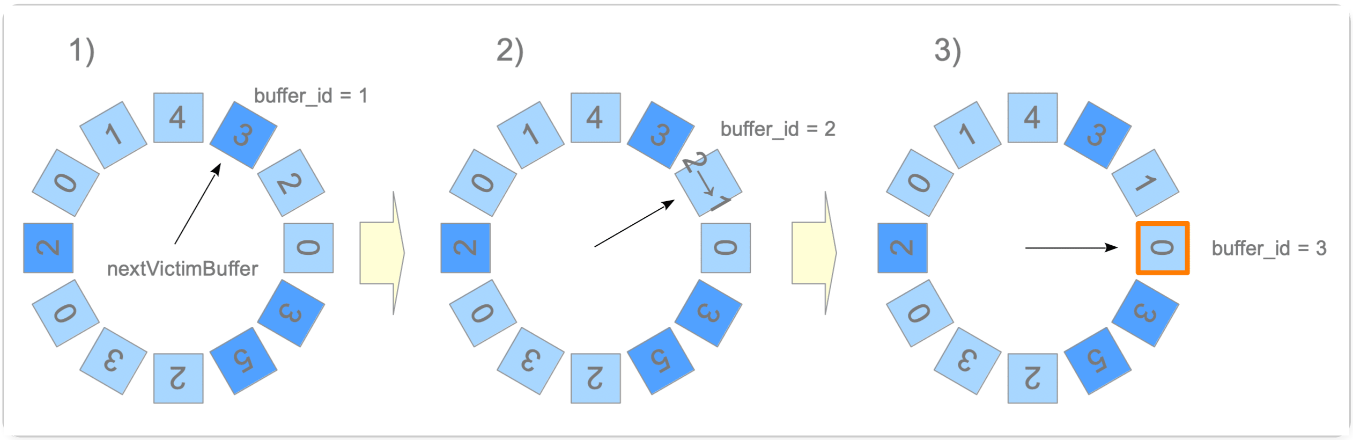
(3) If the *usage_count* of the candidate descriptor is *0*, select the corresponding slot of this descriptor as a victim and proceed to step (5). Otherwise, decrease this descriptor's *usage_count* by 1 and proceed to step (4).

(4) Advance the nextVictimBuffer to the next descriptor (if at the end, wrap around) and return to step (1). Repeat until a victim is found.

(5) Return the buffer_id of the victim.

A specific example is shown in Fig. 8.12. The buffer descriptors are shown as blue or cyan boxes, and the numbers in the boxes show the usage_count of each descriptor.

Fig. 8.12. Clock Sweep.



1) The nextVictimBuffer points to the first descriptor (buffer_id 1). However, this descriptor is skipped because it is pinned.

2) The nextVictimBuffer points to the second descriptor (buffer_id 2). This descriptor is unpinned but its usage_count is 2. Thus, the usage_count is decreased by 1, and the nextVictimBuffer advances to the third candidate.

3) The nextVictimBuffer points to the third descriptor (buffer_id 3). This descriptor is unpinned and its usage_count is 0. Thus, this is the victim in this round.

Whenever the *nextVictimBuffer* sweeps an unpinned descriptor, its *usage_count* is decreased by 1. Therefore, if unpinned descripters exist in the buffer pool, this algorithm can always find a victim, whose usage_count is 0, by rotating the *nextVictimBuffer*.

8.5. Ring Buffer

When reading or writing a huge table, PostgreSQL uses a **ring buffer** instead of the buffer pool. The *ring buffer* is a small and temporary buffer area. When any of the following conditions is met, a ring buffer is allocated to shared memory:

1. Bulk-reading:
   When a relation whose size exceeds one-quarter of the buffer pool size (shared_buffers / 4) is scanned. In this case, the ring buffer size is *256 KB*.
2. Bulk-writing:
   When the SQL commands listed below are executed. In this case, the ring buffer size is *16 MB*.
   - *COPY FROM* command.
   - *CREATE TABLE AS* command.
   - *CREATE MATERIALIZED VIEW* or *REFRESH MATERIALIZED VIEW* command.
   - *ALTER TABLE* command.
3. Vacuum-processing:
   When an autovacuum performs a vacuum processing. In this case, the ring buffer size is *256 KB*.

The allocated ring buffer is released immediately after use.

The benefit of the ring buffer is obvious. If a backend process reads a huge table without using a ring buffer, all stored pages in the buffer pool are evicted, which decreases the cache hit ratio. The ring buffer avoids this issue by providing a temporary buffer area for the huge table.

Why the default ring buffer size for bulk-reading and vacuum processing is 256 KB?

Why 256 KB? The answer is explained in the README located under the buffer manager's source directory.

> For sequential scans, a 256 KB ring is used. That's small enough to fit in L2 cache, which makes transferring pages from OS cache to shared buffer cache efficient. Even less would often be enough, but the ring must be big enough to accommodate all pages in the scan that are pinned concurrently. (snip)

8.6. Flushing Dirty Pages

In addition to replacing victim pages, the checkpointer and background writer processes flush dirty pages to storage. Both processes have the same function, flushing dirty pages, but they have different roles and behaviors.

The checkpointer process writes a checkpoint record to the WAL segment file and flushes dirty pages whenever checkpointing starts. Section 9.7 describes checkpointing and when it begins.

The role of the background writer is to reduce the impact of the intensive writing of checkpointing. The background writer continues to flush dirty pages little by little with minimal impact on database activity. By default, the background writer wakes every 200 msec (defined by bgwriter_delay) and flushes bgwriter_lru_maxpages (the default is 100 pages) at most.

Why the checkpointer was separated from the background writer?

In versions 9.1 or earlier, background writer had regularly done the checkpoint processing. In version 9.2, the checkpointer process has been separated from the background writer process. Since the reason is described in the proposal whose title is "Separating bgwriter and checkpointer", the sentences from it are shown in the following.

> Currently(in 2011) the bgwriter process performs both background writing, checkpointing and some other duties. This means that we can't perform the final checkpoint fsync without stopping background writing, so there is a negative performance effect from doing both things in one process.
>
> Additionally, our aim in 9.2 is to replace polling loops with latches for power reduction. The complexity of the bgwriter loops is high and it seems unlikely to come up with a clean approach using latches.
>
> (snip)