

The Internals of PostgreSQL

for database administrators and system
developers

Chapter 8

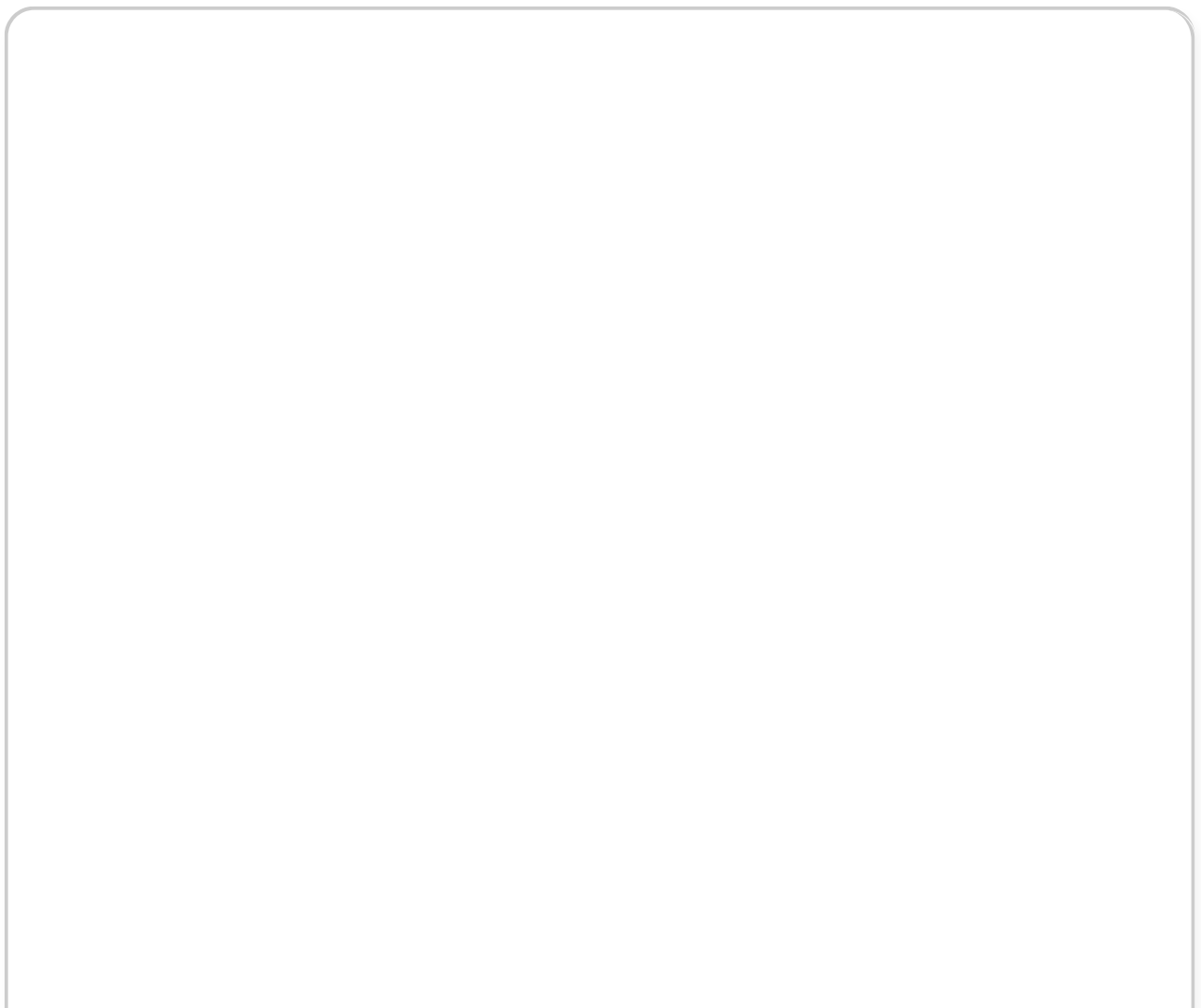
Buffer Manager

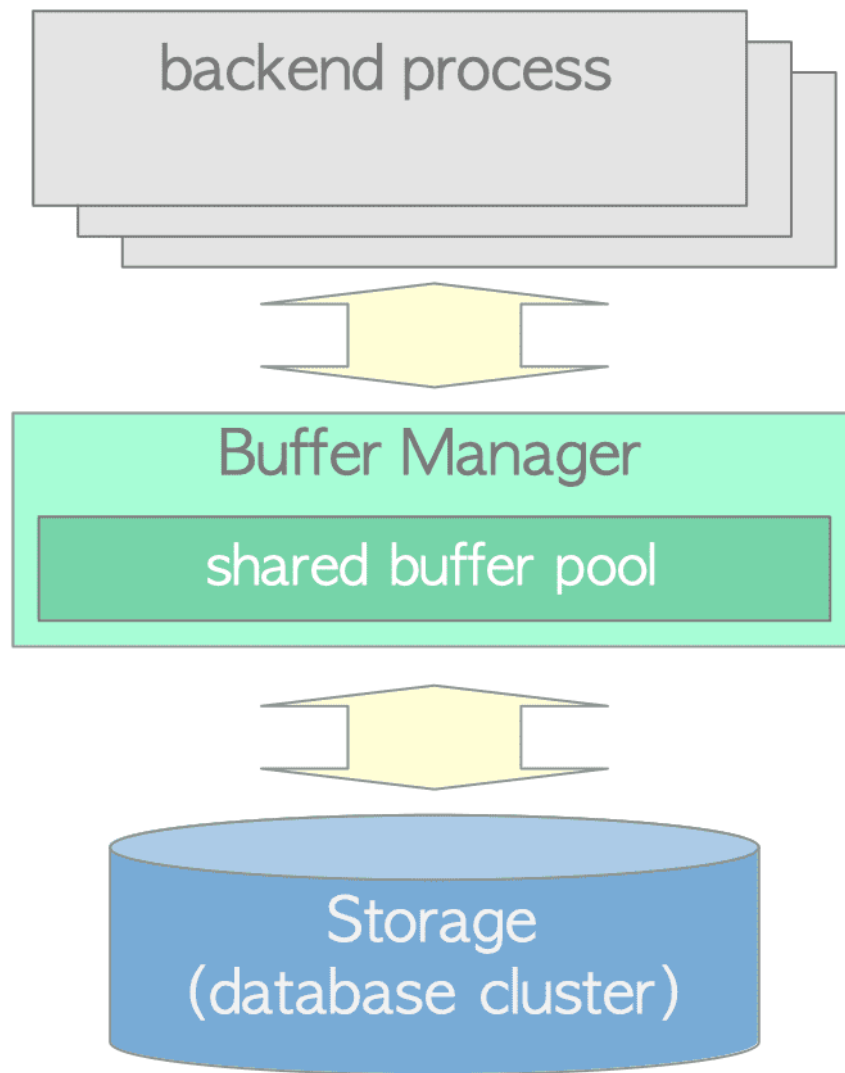
A buffer manager manages data transfers between shared memory and persistent storage and can have a significant impact on the performance of the DBMS. The PostgreSQL buffer manager works very efficiently.

In this chapter, the PostgreSQL buffer manager is described. 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 required to facilitate 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, i.e., each slot stores one page of a data file. Indices of a buffer pool array are referred to as **buffer_ids**.

[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` comprises three values: the [RelFileNode](#) and the fork number of the relation to which its page belongs, and the block number of its page. The fork numbers of tables, freespace maps and visibility maps are defined in 0, 1 and 2, respectively.

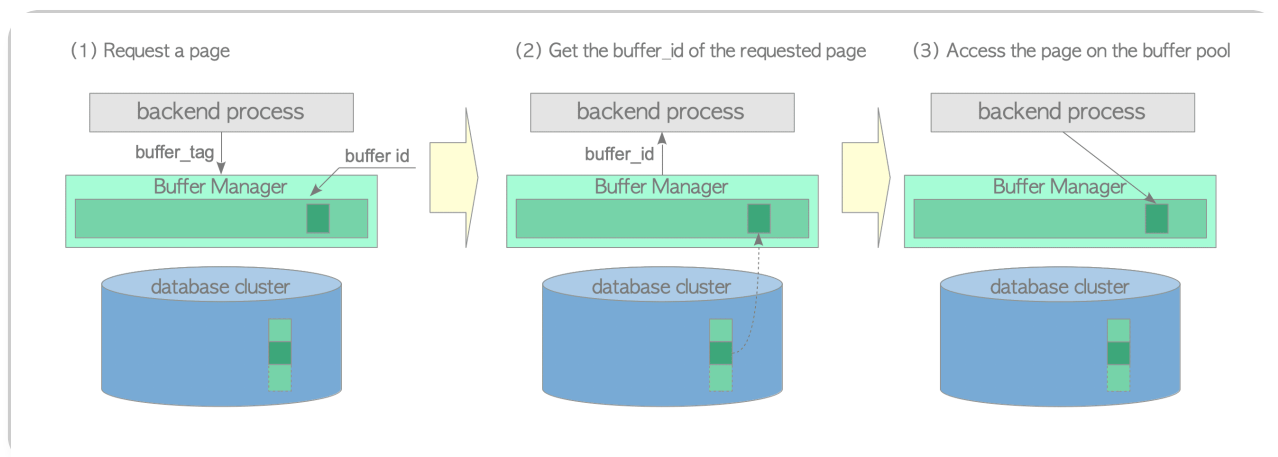
For example, the `buffer_tag` '{(16821, 16384, 37721), 0, 7}' identifies the page that is in the seventh block whose relation's OID and fork number are 37721 and 0, respectively; the relation is contained in the database whose OID is 16384 under the

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's 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 buffer manager works.

8.1.4. Page Replacement Algorithm

When all buffer pool slots are occupied but the requested page is not stored, the buffer manager must select one page in the buffer pool that will 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; thus, many replacement algorithms have

been proposed previously. Since version 8.1, PostgreSQL has used **clock sweep** because it 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.

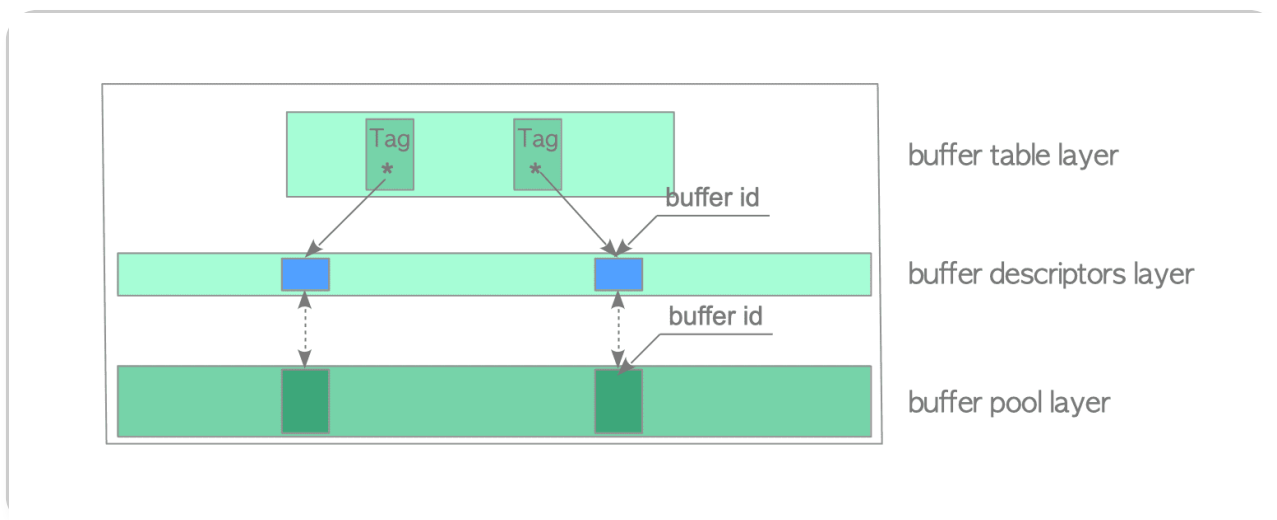
Direct I/O

PostgreSQL does **not** support direct I/O, though sometimes it has been discussed. If you want to know more details, refer to [this discussion](#) on the postgresql-ML and [this article](#).

8.2. Buffer Manager Structure

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

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



- The **buffer pool** is an array. Each slot stores a data file page. The indices of the array slots are referred to as *buffer_ids*.
 - The **buffer descriptors** layer is an array of buffer descriptors. Each descriptor has one-to-one correspondence to a buffer pool slot and holds metadata of the stored page in the corresponding slot.
- Note that the term 'buffer descriptors layer' has

been adopted for convenience and it is only used in this document.

- The **buffer table** is a hash table that stores the relations between the *buffer_tags* of stored pages and the *buffer_ids* of the descriptors that hold the stored pages' respective metadata.

These layers are described in detail in the following subsections.

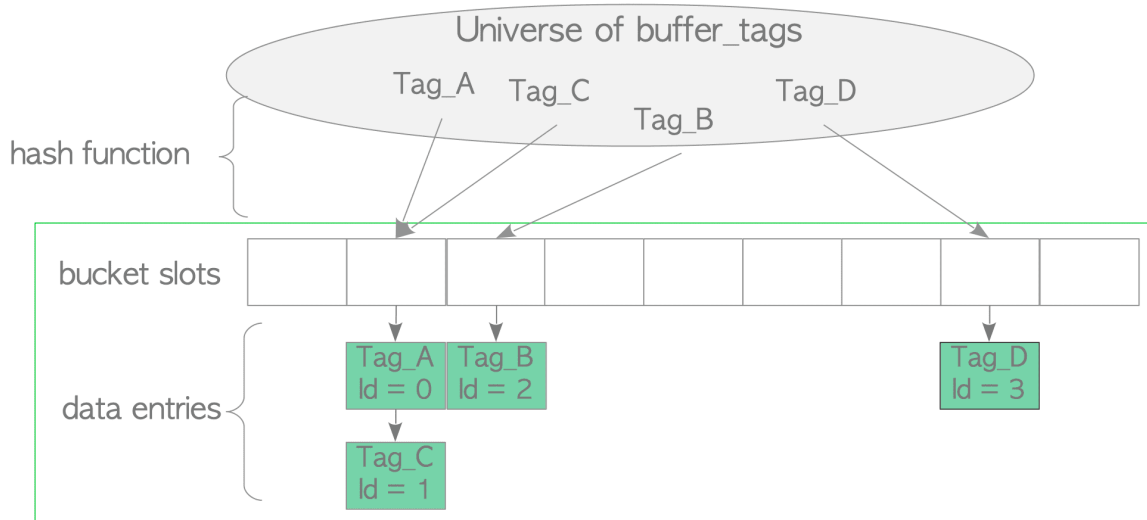
8.2.1. Buffer Table

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 the 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*.

i 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 that basic operations (look up, 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 descriptor is described in this subsection, and the buffer descriptors layer in the next subsection.

Buffer descriptor holds the metadata of the stored page in the corresponding buffer pool slot. The buffer descriptor structure is defined by the structure [BufferDesc](#). While this structure has many fields, mainly ones are shown in the following:

- **tag** holds the *buffer_tag* of the stored page in the corresponding buffer pool slot (buffer tag is defined in [Section 8.1.2](#)).
- **buffer_id** identifies the descriptor (equivalent to the *buffer_id* of the corresponding buffer pool slot).
- **refcount** holds the number of PostgreSQL processes currently accessing the associated stored page. It is also referred to as **pin count**. When a PostgreSQL process accesses the

stored page, its refcount must be incremented by 1 (refcount++). After accessing the page, its refcount must be decreased by 1 (refcount--). When the refcount is zero, i.e. the associated stored page is not currently being accessed, the page is **unpinned**; 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. Note that usage_count is used in the page replacement algorithm ([Section 8.4.4](#)).
- **context_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 whether the stored page is dirty.
 - **valid bit** indicates whether the stored page can be read or written (valid). For example, if this bit is *valid*, then the corresponding buffer pool slot stores a page and this descriptor (valid bit) holds the page metadata; thus, the stored page can be read or written. If this bit is *invalid*, then this

descriptor does not hold any metadata; this means that the stored page cannot be read or written or the buffer manager is replacing the stored page.

- **io_in_progress bit** indicates whether the buffer manager is reading/writing the associated page from/to storage. In other words, this bit indicates whether a single process holds the `io_in_progress_lock` of this descriptor.
- **freeNext** is a pointer to the next descriptor to generate a *freelist*, which is described in the next subsection.



The structure `BufferDesc` is defined in [src/include/storage/buf_internals.h](#).

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 relative to particular conditions, which are described in the next subsection.

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

	(white) <i>Empty</i>
	(blue) <i>Pinned</i>
	(aqua blue) <i>Unpinned</i>

In addition, a dirty page is denoted as 'X'. For example, an unpinned dirty descriptor is represented by .

8.2.3. Buffer Descriptors Layer

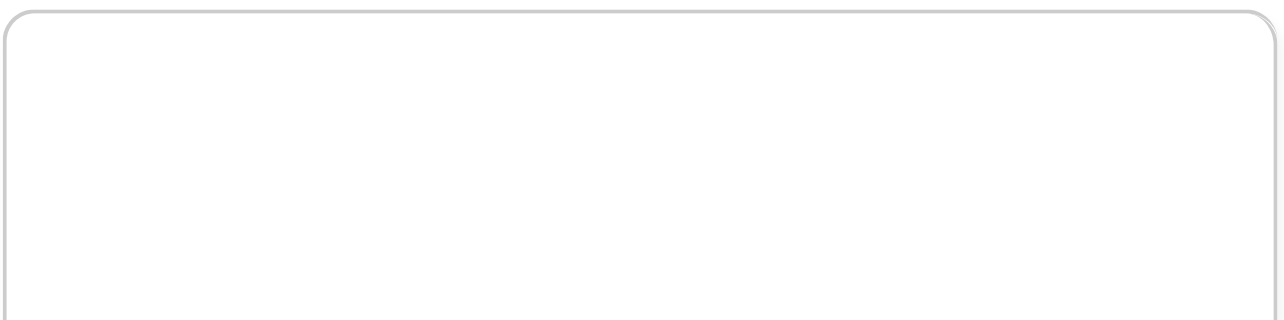
A collection of buffer descriptors forms an array. In this document, the array is referred to as the *buffer descriptors layer*.

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).



Please note that the **freelist** in PostgreSQL is completely different concept from the *freelists* in Oracle. PostgreSQL's freelist is only linked list of empty buffer descriptors. In PostgreSQL *freespace maps*, which are described in [Section 5.3.4](#), act as the same role of the freelists in Oracle.

Fig. 8.5. Buffer manager initial state.



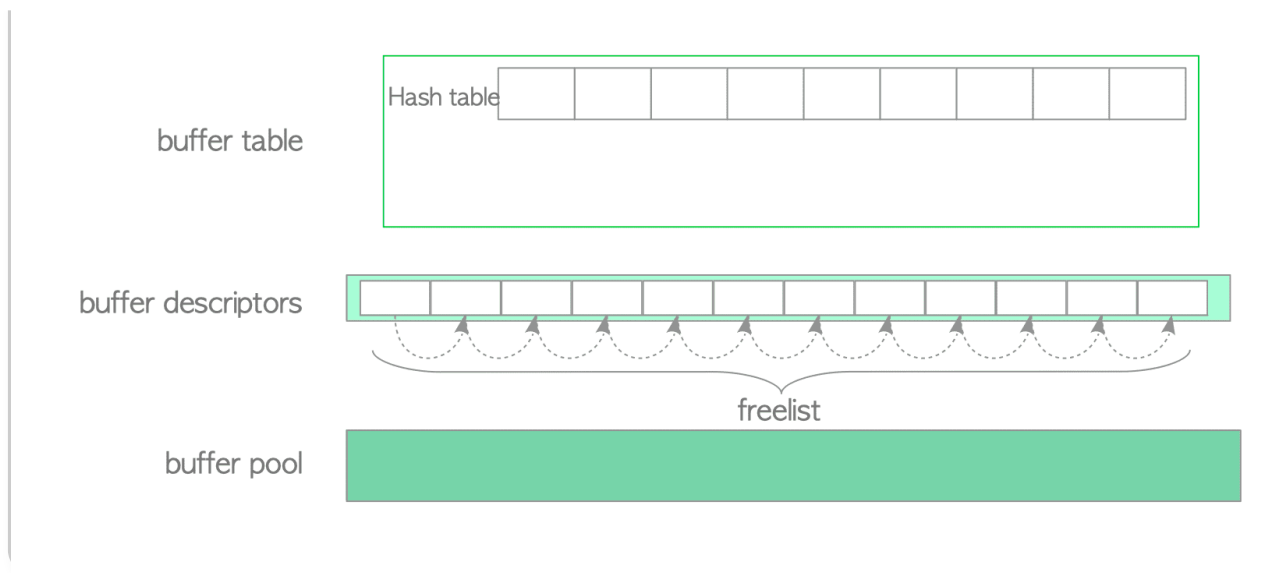
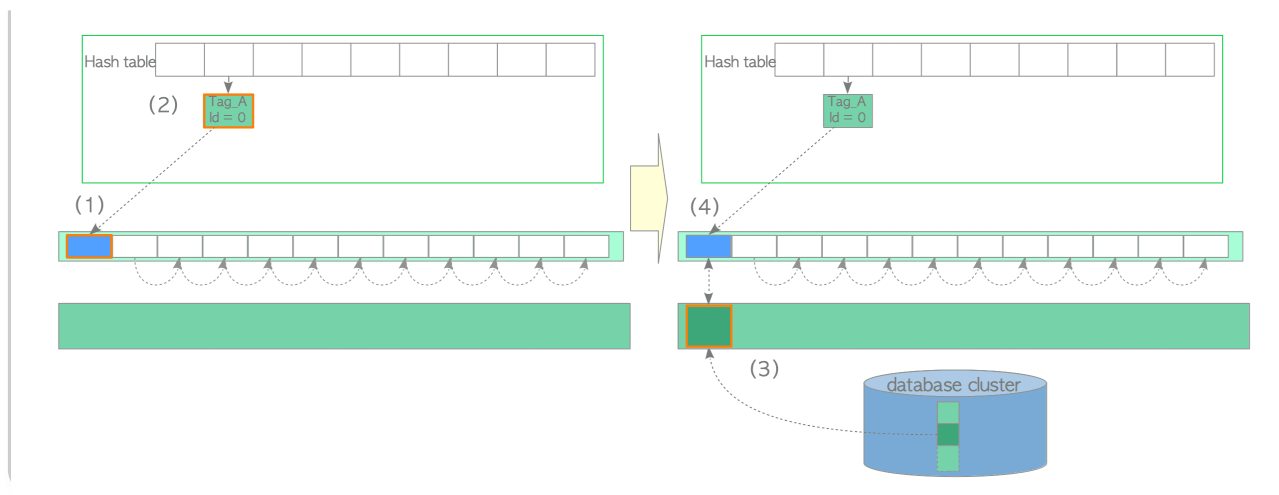


Figure 8.6 shows that how the first page is loaded.

- (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 the new entry, which holds the relation between the tag of the first page and the buffer_id of the retrieved descriptor, in the buffer table.
- (3) Load the new page from storage to 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 continue to be used do not return to the freelist. However, related descriptors are added to the freelist again and the descriptor state becomes 'empty' when one of the following occurs:

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

i Why empty descriptors comprise the freelist?

The reason why the freelist be made is to get the first descriptor immediately. This is a usual practice for dynamic memory resource allocation. 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. Indices of the buffer pool array are referred to as *buffer_ids*.

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

8.3. Buffer Manager Locks

The buffer manager uses many locks for many different purposes. This section describes the locks necessary for the explanations in the subsequent sections.



Please note that the locks described in this section are parts of a synchronization mechanism for the

buffer manager; they do **not** relate to any SQL statements and SQL options.

8.3.1. Buffer Table Locks

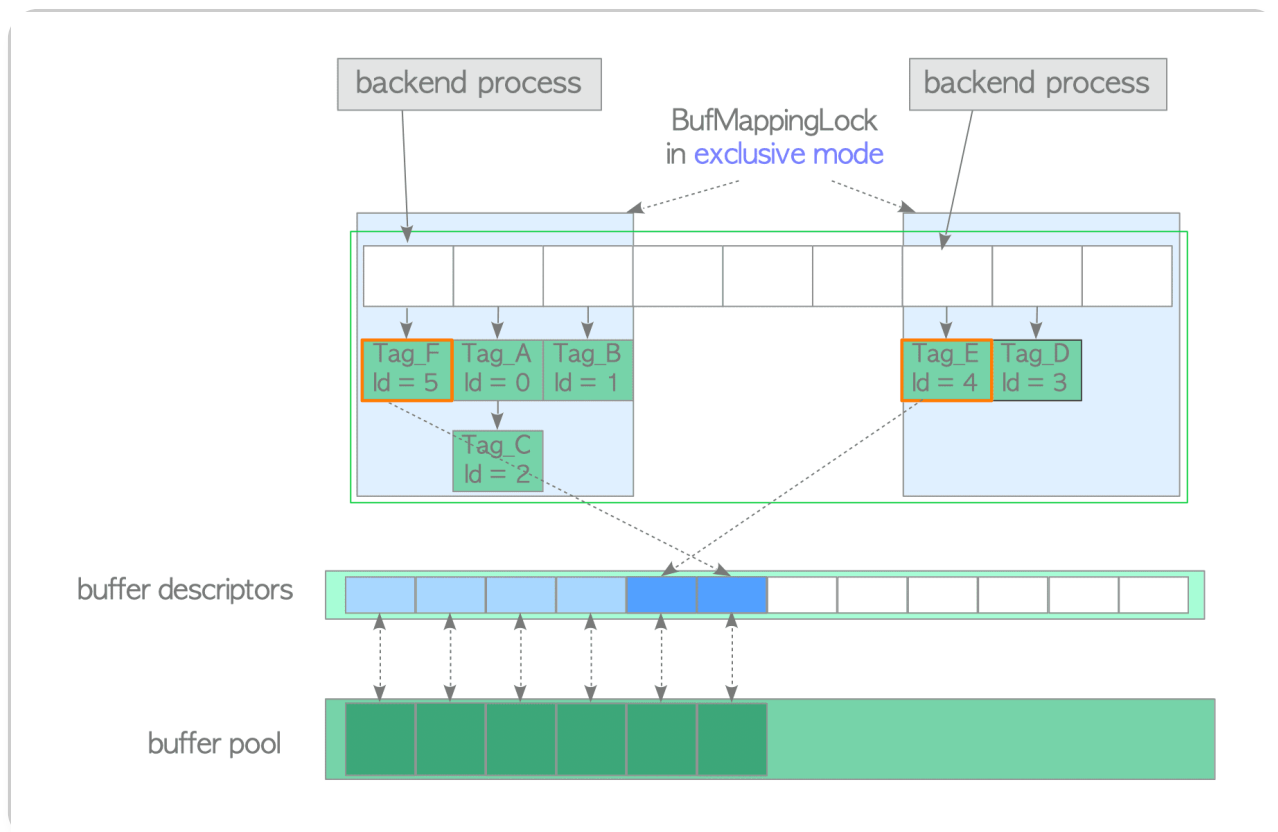
BufMappingLock protects the data integrity of the entire buffer table. It is a light-weight lock that can be used in both shared and exclusive modes. When searching an entry in the buffer table, a backend process holds a shared BufMappingLock. When inserting or deleting entries, a backend process holds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

The BufMappingLock is split into partitions to reduce the contention in the buffer table (the default is 128 partitions). Each BufMappingLock partition guards the 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 in order to insert new data entries. If the BufMappingLock is a single system-wide lock, both

processes should wait for the processing of another process, depending on which started processing.

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.

The BufMappingLock had been split into 16 separate locks by default until version 9.4.

8.3.2. Locks for Each Buffer Descriptor

Each buffer descriptor uses two light-weight locks, **content_lock** and **io_in_progress_lock**, to control access to the stored page in the corresponding buffer pool slot. When the values of own fields are checked or changed, a spinlock is used.

8.3.2.1. content_lock

The `content_lock` is a typical lock that enforces access limits. 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.

However, 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).

- Removing tuples physically or compacting free space on the stored page (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 shows more details.

8.3.2.2. io_in_progress_lock

The `io_in_progress` lock is used to wait for I/O on a buffer to complete. When a PostgreSQL process loads/writes page data from/to storage, the process holds an exclusive `io_in_progress` lock of the corresponding descriptor while accessing the storage.

8.3.2.3. spinlock

When the flags or other fields (e.g. `refcount` and `usage_count`) are checked or changed, a spinlock is used. Two specific examples of spinlock usage are given below:

(1) The following shows how to **pin** the 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->refcount++;
bufferdesc->usage_count++;
UnlockBufHdr(bufferdesc); /* Release the
    spinlock */
```

(2) The following shows how to set 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.

i Replacing buffer manager spinlock with atomic operations

In version 9.6, the spinlocks of buffer manager will be replaced to atomic operations. See this [result of commitfest](#). If you want to know the details, refer to [this discussion](#).

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.

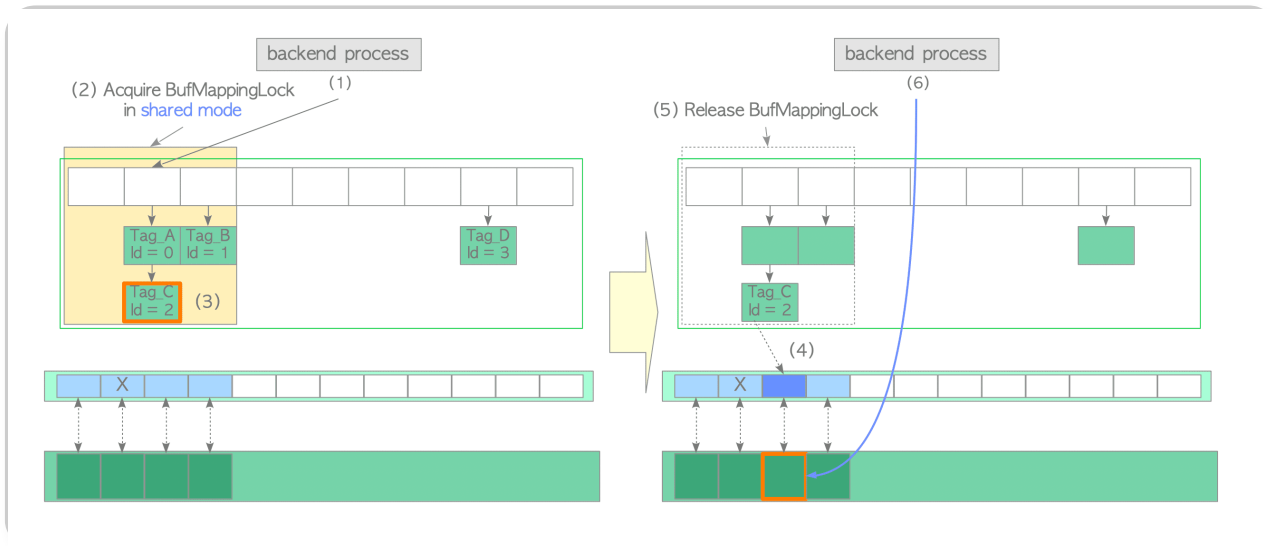
8.4.1. Accessing a Page Stored in the Buffer Pool

First, the simplest case is described, i.e. 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*, which 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, i.e. the *refcount* and *usage_count* of the descriptor are increased 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. Thus, 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.

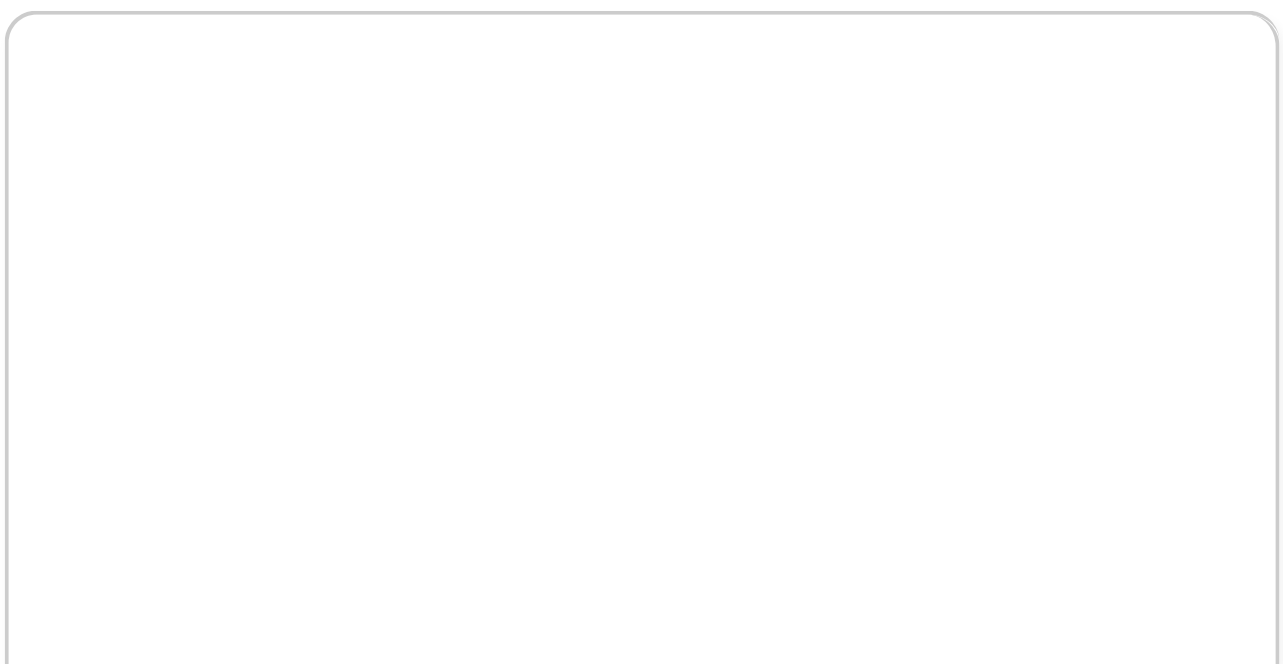
8.4.2. Loading a Page from Storage to Empty Slot

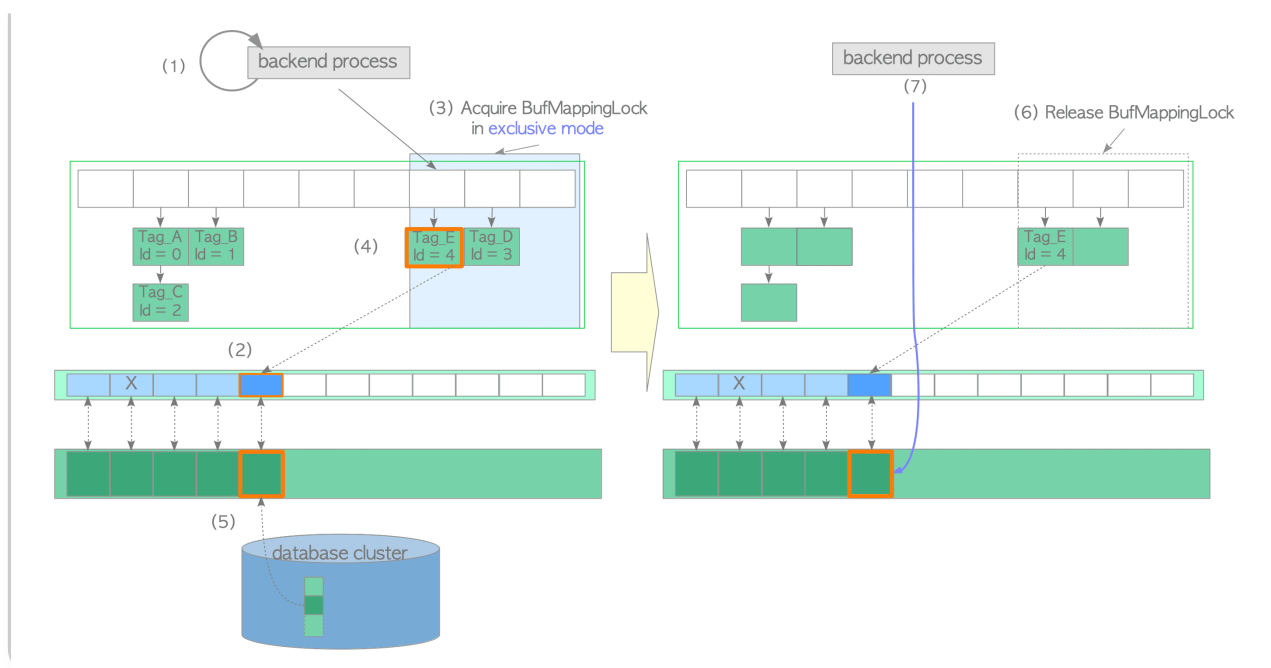
In this second case, assume that the desired page is not in the buffer pool and the freelist has free elements (empty descriptors). In this case, the buffer manager performs the following steps:

- (1) Look up the buffer table (we assume 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. 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. 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](#).

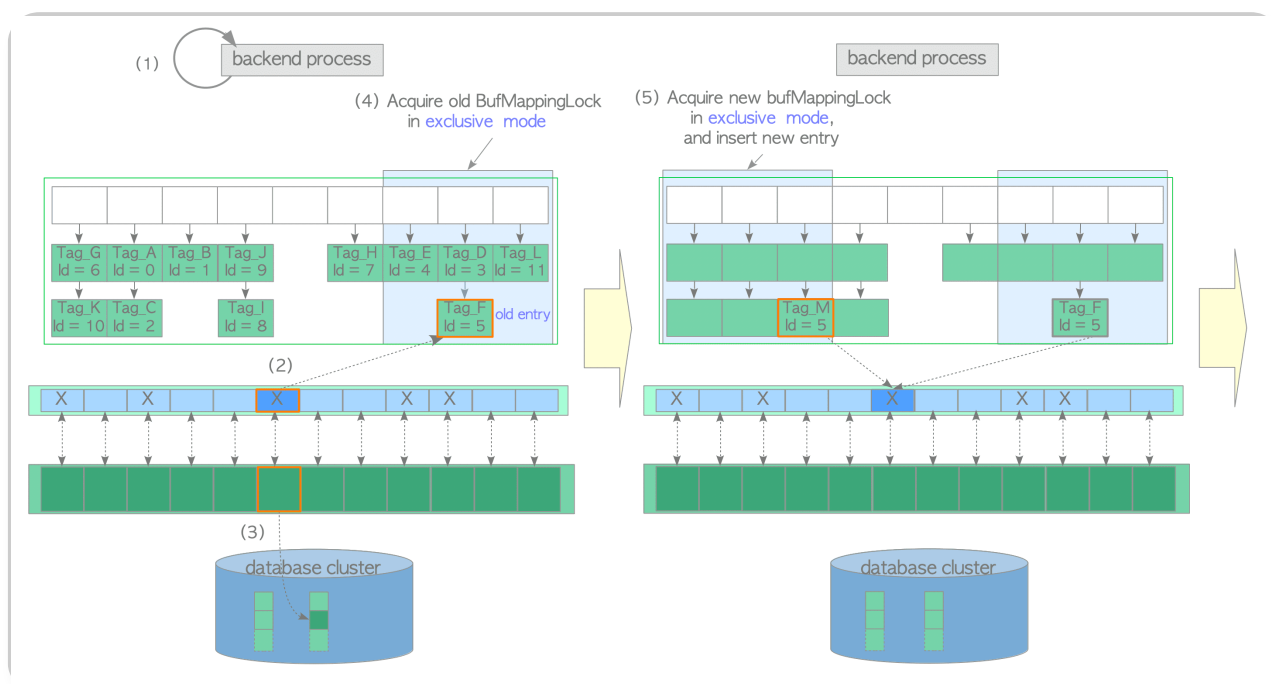
(3) Flush (write and `fsync`) the victim page data if it is dirty; otherwise proceed to step (4).

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.

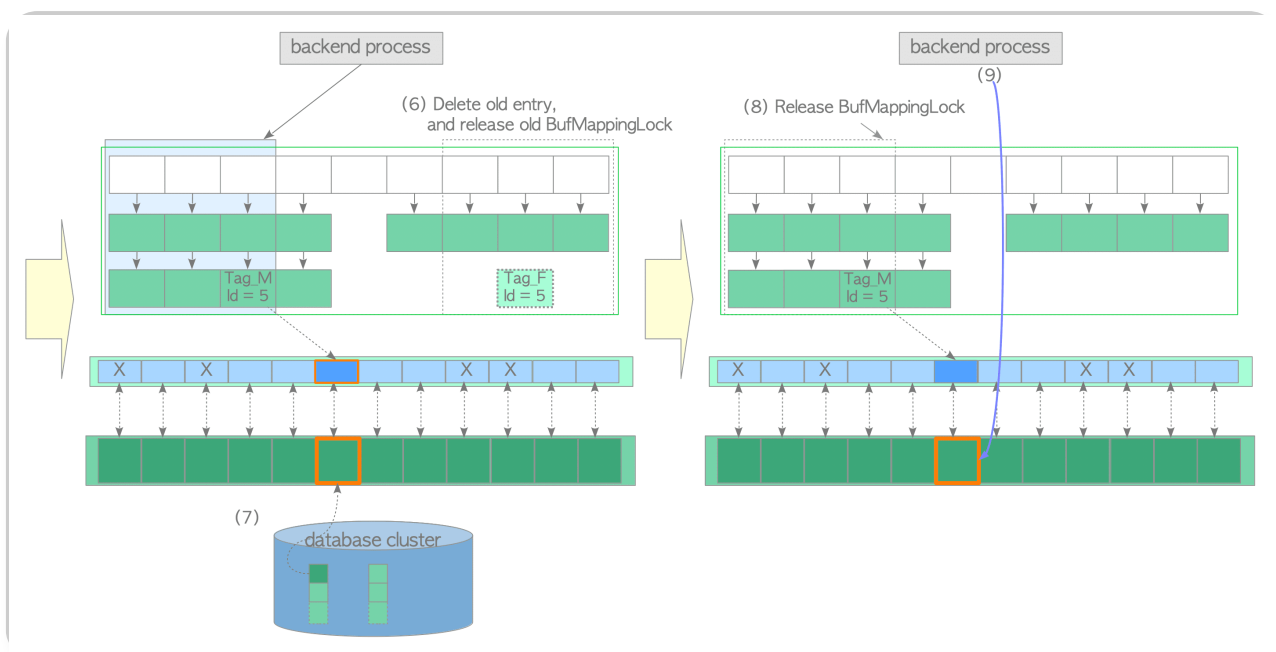
Fig. 8.10. Loading a page from storage to a victim buffer pool slot.



- (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 initialize other bits.
- (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

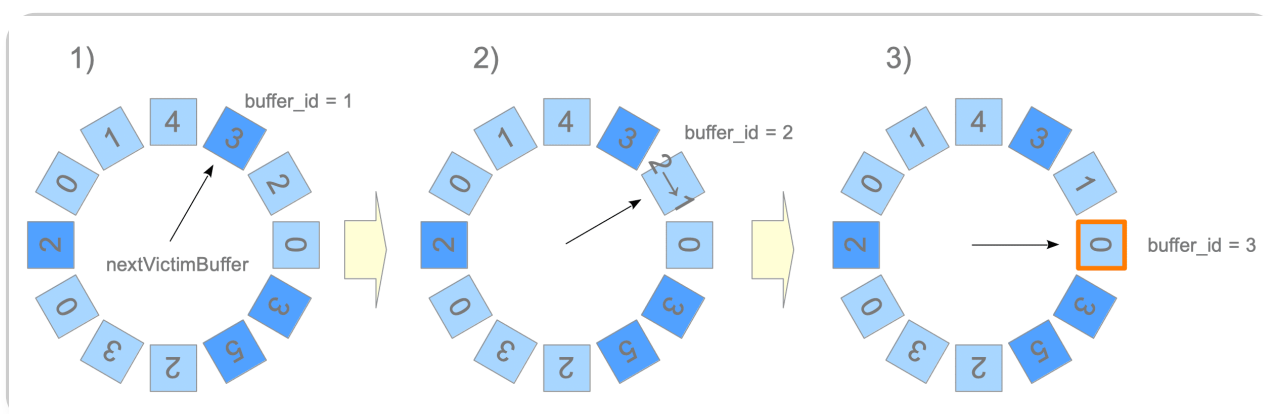
```
WHILE true
(1)  Obtain the candidate buffer descriptor pointed by the nextVictimBuffer
(2)  IF the candidate descriptor is unpinned THEN
(3)    IF the candidate descriptor's usage_count == 0 THEN
        BREAK WHILE LOOP /* the corresponding slot of this descriptor is victim slot. */
    ELSE
  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

- (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 descriptors 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** rather than the buffer pool. The *ring buffer* is a small and temporary buffer area. When any condition listed below 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 ($\text{shared_buffers}/4$) 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 removed (kicked out); therefore, the cache hit ratio decreases. The ring buffer avoids this issue.

i 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); however, they have different roles and behaviours.

The checkpoint 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 influence 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 maximum.

i Why the checkpointer was separated from the background writer?

In version 9.1 or earlier, background writer had regularly done the checkpoint processing. In version 9.2, the checkpoint 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)