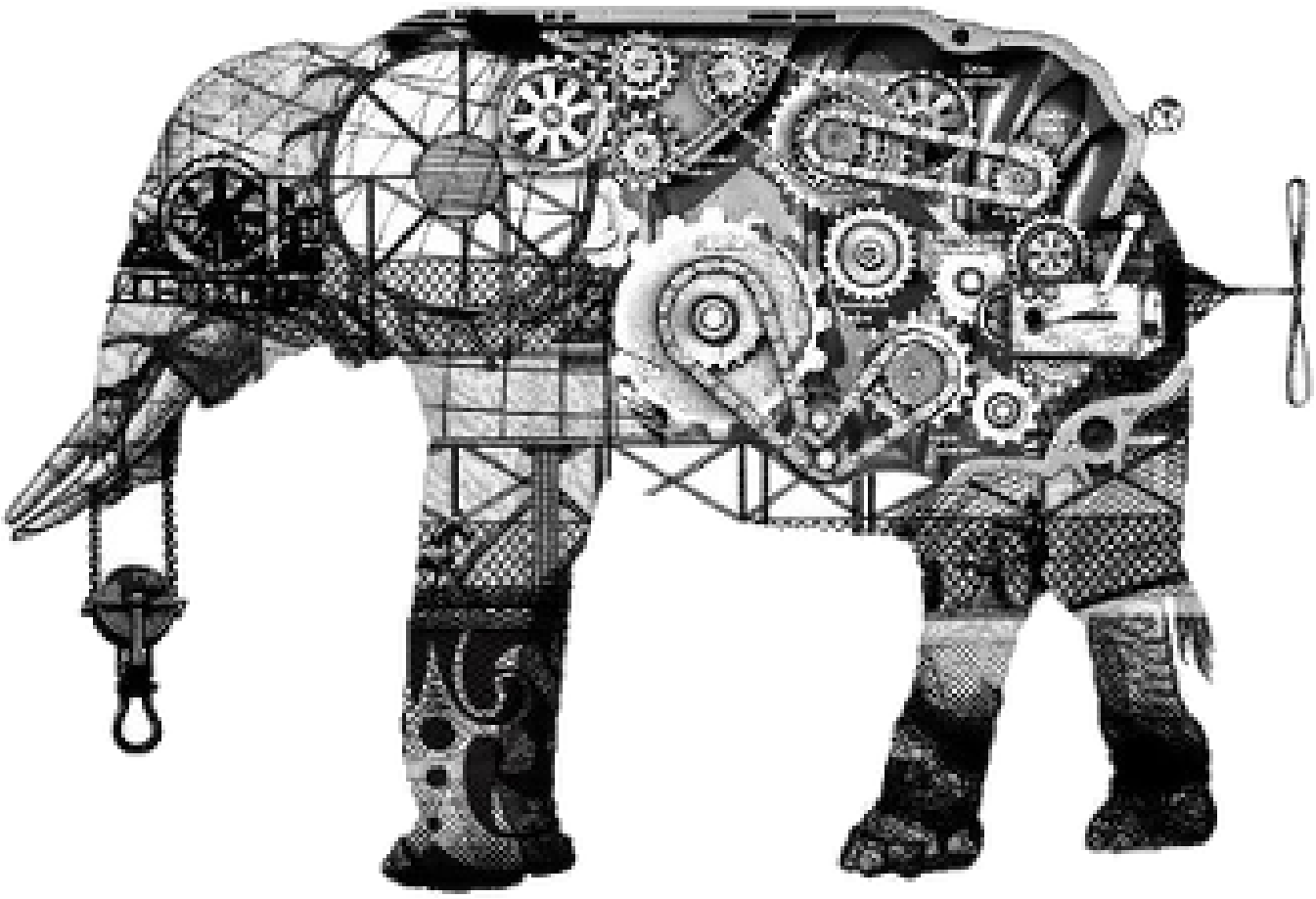
The Internals of PostgreSQL

for database administrators and system developers



Chapter 9

Write Ahead Log(ging) — WAL

A **transaction log** is an essential part of database, because all of the database management system is required not to lose any data even when a system failure occurs. It is a history log of all changes and actions in a database system so as to ensure that no data has been lost due to failures, such as a power failure, or some other server failure that causes the server crash. As the log contains sufficient information about each transaction executed already, the database server should be able to recover the database cluster by replaying changes and actions in the transaction log in case of the server crash.

In the field of computer science, **WAL** is an acronym of **Write Ahead Logging**, which is a protocol or a rule to write both changes and actions into a transaction log, whereas in PostgreSQL, WAL is an acronym of **Write Ahead Log**. There the term is used as synonym of transaction log, and also used to refer to an implemented mechanism related to writing action to a transaction log (WAL). Though this is a little confusing, in this document the PostgreSQL definition has been adopted.

The WAL mechanism was first implemented in version 7.1 to mitigate the impacts of server crashes. It also made possible the implementation of the Point-in-Time Recovery (PITR) and Streaming Replication (SR), both of which are described in Chapter 10 and Chapter 11 respectively.

Although understanding of the WAL mechanism is essential for system integrations and administrations using PostgreSQL, the complexity of this mechanism makes it impossible to summarize its description in brief. So the complete explanation of WAL in PostgreSQL is made as follows. In the first section, the overall picture of the WAL has been provided, introducing some important concepts and keywords. In the subsequent sections, the following topics are described:

- The logical and physical structures of the WAL (transaction log)
- The internal layout of WAL data
- Writing of WAL data
- WAL writer process
- The checkpoint processing
- The database recovery processing
- Managing WAL segment files
- Continuous archiving

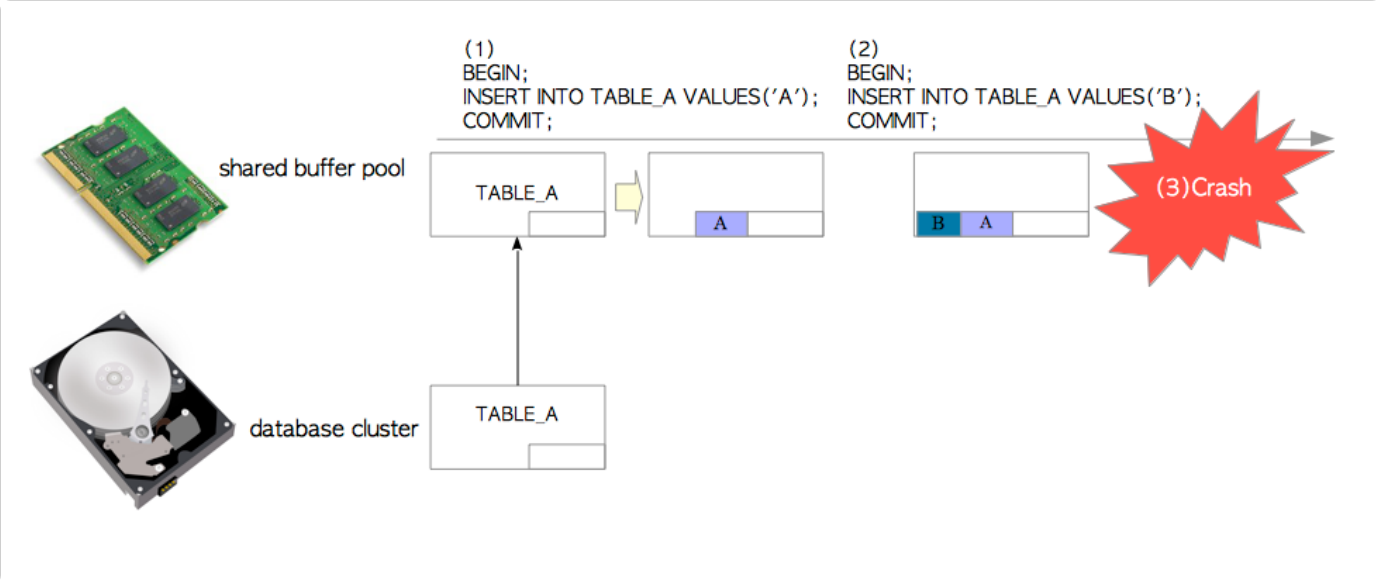
In this section, to simplify the description, the table TABLE_A which contains just one page has been used.

9.1.1 Insertion operations without WAL

As described in Chapter 8, to provide efficient access to the relation's pages, every DBMS implements shared buffer pool.

Assume that we insert some data tuples into TABLE_A on PostgreSQL which does not implement the WAL feature; this situation is illustrated in Figure 9.1.

Figure 9.1: Insertion operations without WAL



(1) Issuing the first INSERT statement, PostgreSQL loads the TABLE_A's page from a database cluster into the in-memory shared buffer pool, and inserts a tuple into the page. This page is not written into the database cluster immediately. (As mentioned in Chapter 8, a modified pages are generally called a **dirty page**.)
(2) Issuing the second INSERT statement, PostgreSQL inserts a new tuple into the page on the buffer pool. This page has not been written into the storage yet.
(3) If the operating system or PostgreSQL server should fail for any reasons such as a power failure, all of the inserted data would be lost.

So, database without WAL is vulnerable to the system failures.

9.1.2 Insertion operations and database recovery

To deal with the system failures mentioned above without compromising performance, PostgreSQL supports the WAL. In this subsection, some keywords and key concepts, and then the writing of WAL data and the recovering of the database are described.

PostgreSQL writes all modifications as history data into a persistent storage, to prepare for failures. In PostgreSQL, the history data are known as **XLOG record(s)** or **WAL data**.

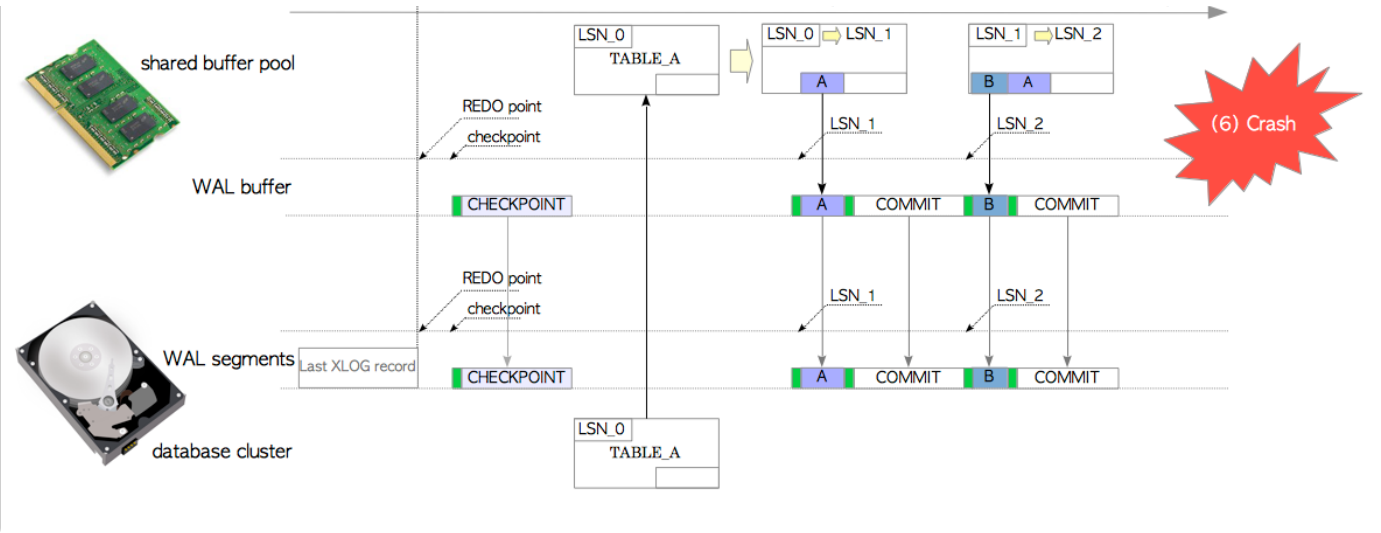
XLOG records are written into the in-memory **WAL buffer** by change operations such as insertion, deletion, or commit action. They are immediately written into a **WAL segment file** on the storage when a transaction commits/aborts. (To be precise, the writing of XLOG records may occur in other cases. The details will be described in Section 9.5.) **LSN (Log Sequence Number)** of XLOG record represents the location where its record is written on the transaction log. LSN of record is used as the unique id of XLOG record.

By the way, when we consider how database system recovers, there may be one question; what point does PostgreSQL start to recover from? The answer is **REDO point**; that is, the location to write the XLOG record at the moment when the latest **checkpoint** is started (checkpoint in PostgreSQL is described in Section 9.7). In fact, the database recovery processing is strongly linked to the *checkpoint processing* and both of these processing are inseparable.

The WAL and checkpoint process were implemented at the same time in version 7.1.

As the introduction of major keywords and concepts has just finished, from now on will be the description of the tuple insertion with WAL. See Figure 9.2 and the following description. (Also refer to this slide.)

Figure 9.2: Insertion operations with WAL



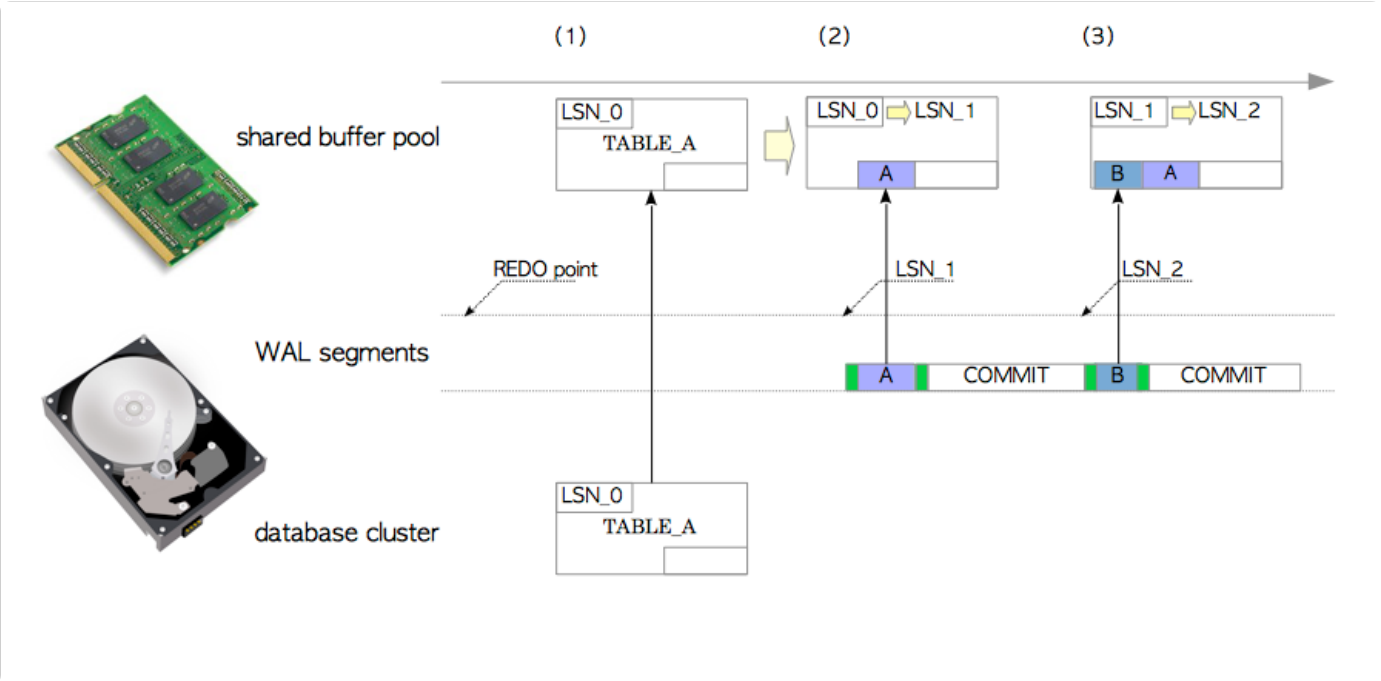
Notation

"*TABLE_A's LSN*" shows the value of pd_lsn within the page-header of TABLE_A. "*page's LSN*" is the same manner.

(1) A checkpointer, a background process, periodically performs checkpointing. Whenever the checkpointer starts, it writes a XLOG record called **checkpoint record** to the current WAL segment. This record contains the location of the latest *REDO point*.
(2) Issuing the first INSERT statement, PostgreSQL loads the TABLE_A's page into the shared buffer pool, inserts a tuple into the page, creates and writes a XLOG record of this statement into the WAL buffer at the location *LSN_1*, and updates the TABLE_A's LSN from *LSN_0* to *LSN_1*.
In this example, this XLOG record is a pair of a header-data and the *tuple entire*.
(3) As this transaction commits, PostgreSQL creates and writes a XLOG record of this commit action into the WAL buffer, and then, writes and flushes all XLOG records on the WAL buffer to the WAL segment file, from *LSN_1*.
(4) Issuing the second INSERT statement, PostgreSQL inserts a new tuple into the page, creates and writes this tuple's XLOG record to the WAL buffer at *LSN_2*, and updates the TABLE_A's LSN from *LSN_1* to *LSN_2*.
(5) When this statement's transaction commits, PostgreSQL operates in the same manner as in step (3).
(6) Imagine when the operating system failure should occur. Even though all of data on the shared buffer pool are lost, all modifications of the page have been written into the WAL segment files as history data.

The following instructions show how to recover our database cluster back to the state immediately before the crash. There is no need to do anything special, since PostgreSQL will automatically enter into the recovery-mode by restarting. See Figure 9.3 (and this slide). PostgreSQL sequentially will read and replay XLOG records within the appropriate WAL segment files from the REDO point.

Figure 9.3: Database recovery using WAL



(1) PostgreSQL reads the XLOG record of the first INSERT statement from the appropriate WAL segment file, loads the TABLE_A's page from the database cluster into the shared buffer pool.
(2) Before trying to replay the XLOG record, PostgreSQL shall compare the XLOG record's LSN with the corresponding page's LSN, the reason why doing this will be described in Section 9.8. The rules of the replaying XLOG records are shown below.

(5) PostgreSQL replays the remaining XLOG record(s) in the same way.

PostgreSQL can recover itself in this way by replaying XLOG records written in WAL segment files in chronological order. Thus, PostgreSQL's XLOG records are obviously **REDO log**.

> ⚠ PostgreSQL does not support UNDO log.

Though writing XLOG records certainly costs a certain amount, it' s nothing compared to writing the entire modified pages. We are sure we can get a greater benefit, i.e., the system failure tolerance, than the amount we paid.

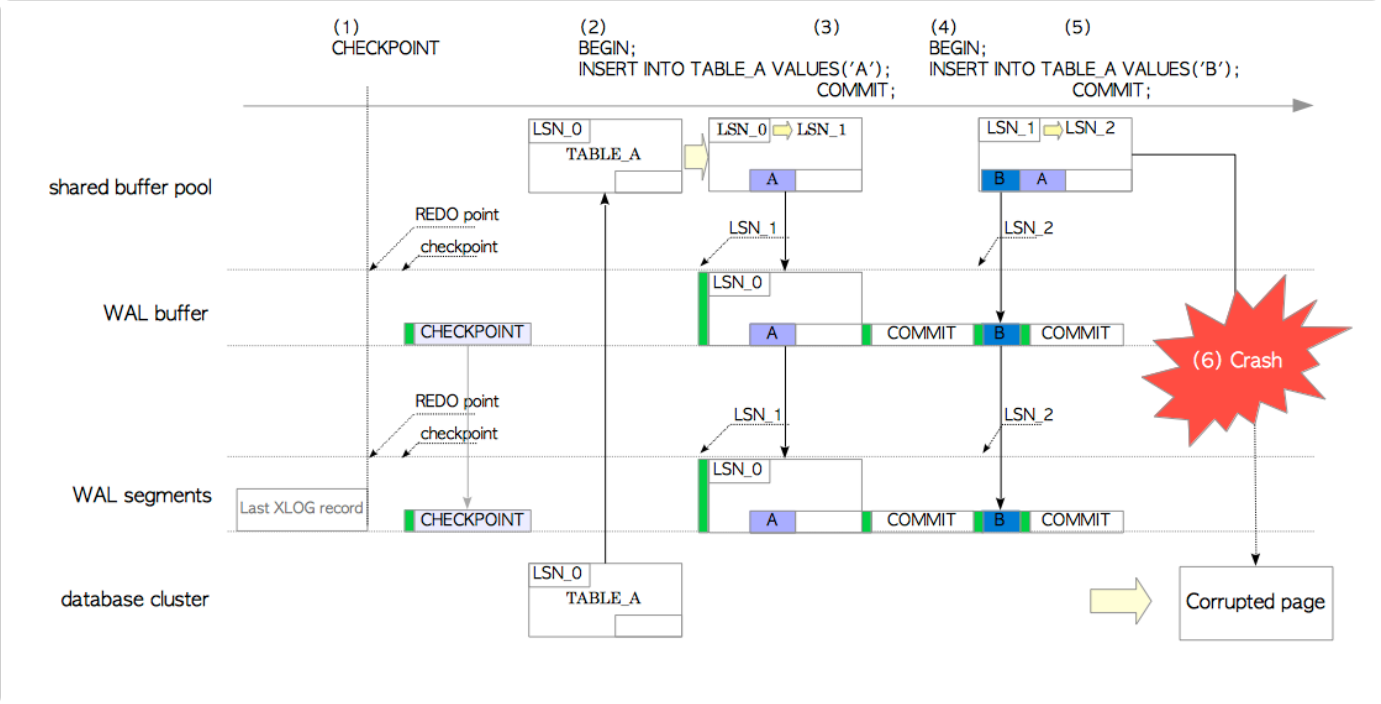
9.1.3 Full-page writes

Suppose that the TABLE_A's page-data on the storage is corrupted, because the operating system has failed while the background-writer process has been writing the dirty pages. As XLOG records cannot be replayed on the corrupted page, we would need an additional feature.

PostgreSQL supports a feature referred to as **full-page writes** to deal with such failures. If it is enabled, PostgreSQL writes a pair of the header-data and the *entire page* as XLOG record during the first change of each page after every checkpoint; default is enabled. In PostgreSQL, such a XLOG record containing the entire page is referred to as **backup block** (or **full-page image**).

Let's describe the insertion of tuples again, but with the full-page-writes enabled. See Figure 9.4 and the following description.

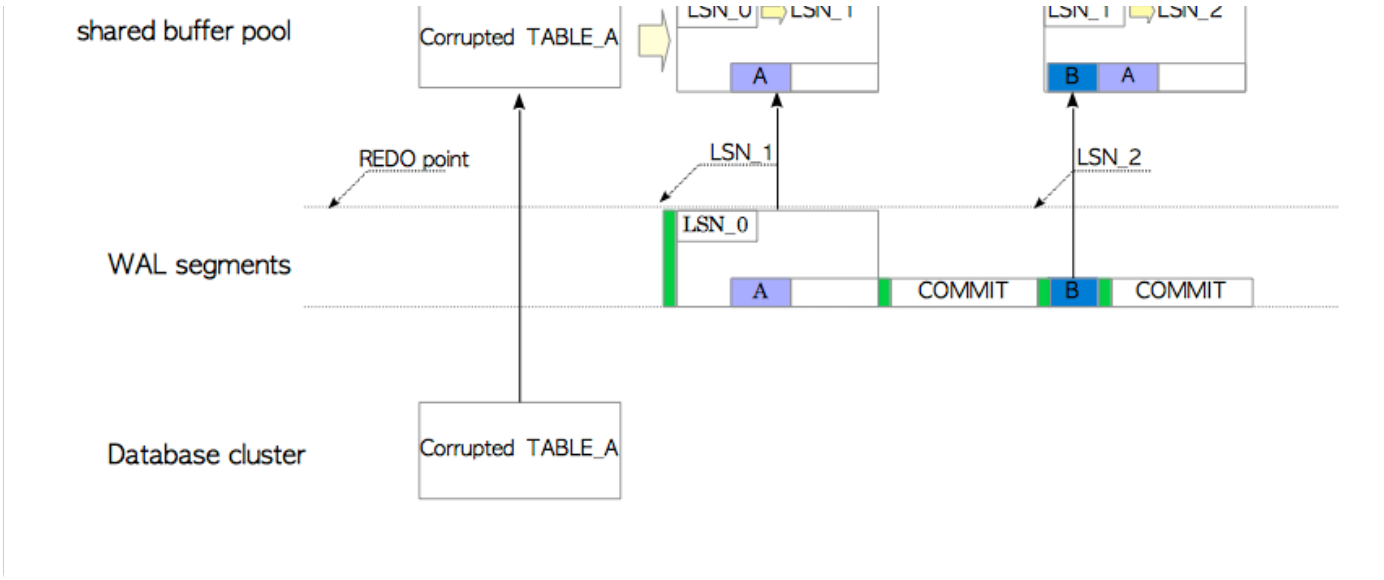
Figure 9.4: Full page writes



(1) The checkpointer starts a checkpoint process.
(2) In the insertion of the first INSERT statement, though PostgreSQL operates in the almost the same manner as in the previous subsection, this XLOG record is the *backup block* of this page (i.e. it contains the page entirety), because this is the first writing of this page after the latest checkpoint.
(3) As this transaction commits, PostgreSQL operates in the same manner as in the previous subsection.
(4) In the insertion of the second INSERT statement, PostgreSQL operates in the same manner as in the previous subsection since this XLOG record is not a backup block.
(5) When this statement's transaction commits, PostgreSQL operates in the same manner as in the previous subsection.
(6) To demonstrate the effectiveness of full-page writes, here we consider the case in which the TABLE_A's page on the storage has been corrupted due to the operating system failure occurred while the background-writer has been writing it into the HDD.

Restart the PostgreSQL server to repair the broken cluster. See Figure 9.5 and the following description.

Figure 9.5: Database recovery with backup block



(1) PostgreSQL reads the XLOG record of the first INSERT statement and loads the corrupted TABLE_A's page from the database cluster into the shared buffer pool. In this example, the XLOG record is a backup block, because the first XLOG record of each page is always its backup block according to the writing rule of full-page writes.
(2) When a XLOG record is its backup block, another rule of replaying is applied: the record's data-portion (i.e. the page itself) is to be overwritten onto the page regardless of the values of both LSNs, and the page's LSN updated to the XLOG record's LSN.
In this example, PostgreSQL overwrites the data-portion of the record to the corrupted page, and updates the TABLE_A's LSN to *LSN_1*. In this way, the corrupted page is restored by its backup block.
(3) Since the second XLOG record is a non-backup block, PostgreSQL operates in the same manner as the instruction in the previous subsection.

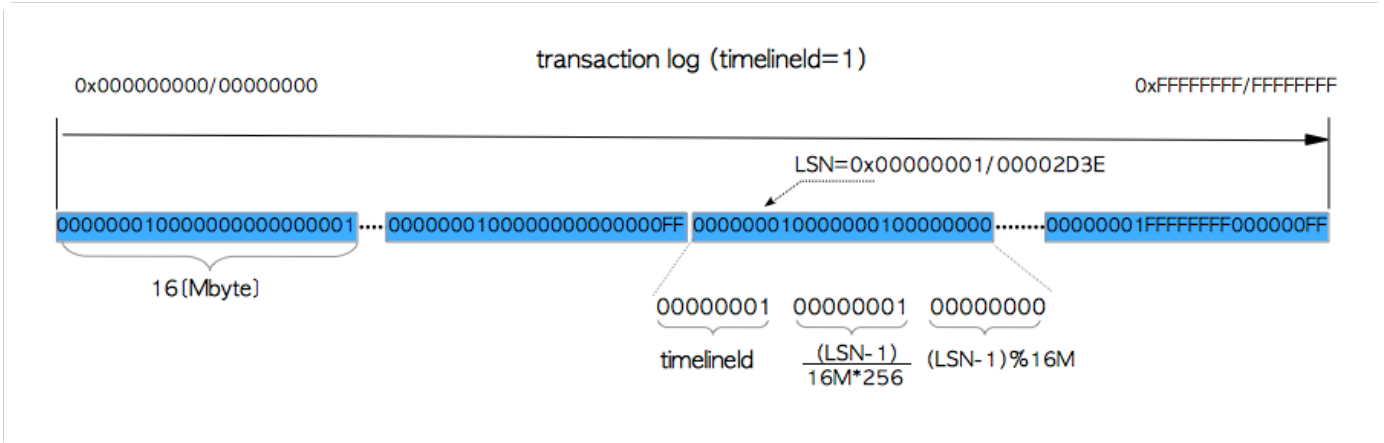
PostgreSQL can be recovered even if some data write failures have occurred. (Of course, this does not apply if the file-system or media failure has occurred.)

9.2 Transaction log and WAL segment files

Logically, PostgreSQL writes XLOG records into the transaction log which is a virtual file 8-byte long (16 ExaByte).

Since a transaction log capacity is effectively unlimited and so can be said that 8-byte address space is vast enough, it is impossible for us to handle a file with the capacity of 8-byte length. So, a transaction log in PostgreSQL is divided into files of 16-Mbyte each of which known as *WAL segment*. See Figure 9.6.

Figure 9.6: Transaction log and WAL segment files



The WAL segment filename is in hexadecimal 24-digit number and the naming rule is as follows:

```
WAL segment file name = timelineId + (uint32)((LSN-1)/(16M*256))  + (uint32)((LSN-1) % 16M)
```

timelineId

PostgreSQL's WAL contains the concept of **timelineId** (4-byte unsigned integer), which is for Point-in-Time Recovery (PITR) described in Chapter 10. However, the timelineId is fixed to 0x00000001 in this chapter because this concept is not required in the following descriptions.

The first WAL segment file is 000000010000000000000001. If the first one has been filled up with the writing of XLOG records, the second one 000000010000000000000002 would be provided. Files of successor is used in ascending order in succession, after 00000001000000000000000FF has been filled up, next one 000000010000000100000000 will be provided. In this way, whenever the last 2-digit carries over, the middle 8-digit number increases one.

Similarly, after 000000010000000100000FF has been filled up, 000000010000000200000000 will be provided, and so on.

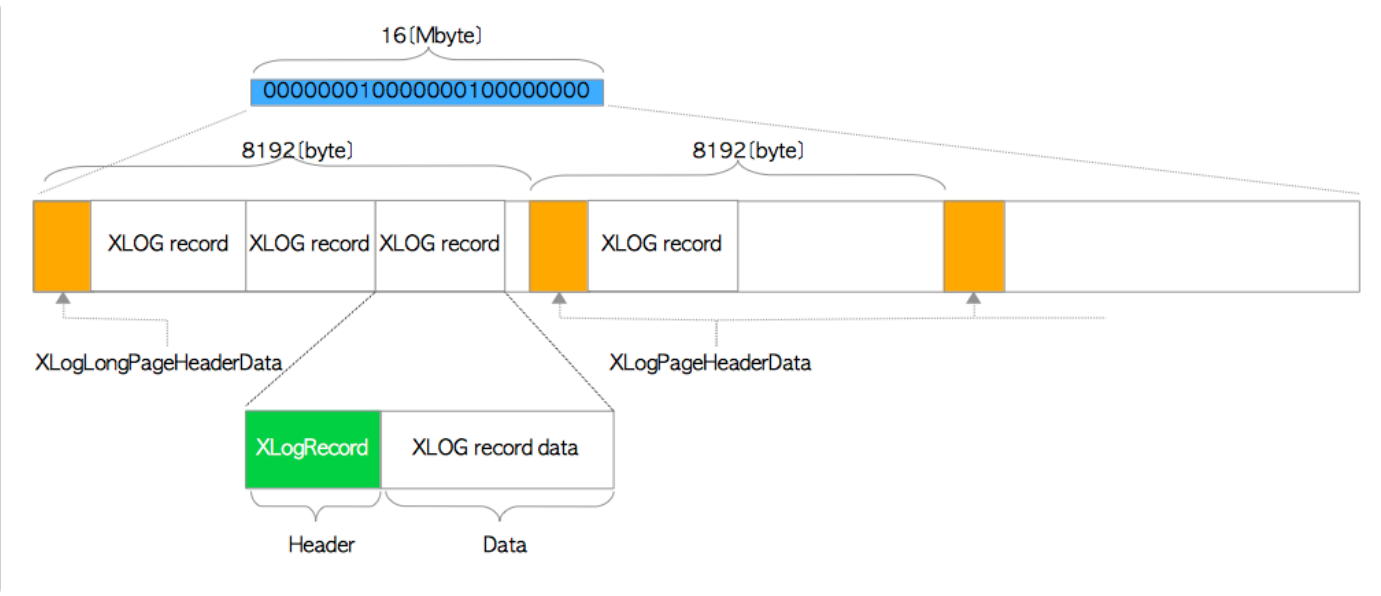
pg_xlogfile_name / pg_walfile_name

```
000000010000000100000000
(1 row)
```

9.3 Internal layout of WAL segment

A WAL segment is a 16 MB file, and it is internally divided into pages of 8192 bytes (8 KB). The first page has a header-data defined by the structure XLogLongPageHeaderData, while the headings of all other pages have the page information defined by the structure XLogPageHeaderData. Following the page header, XLOG records are written in each page from the beginning in descending order. See Figure 9.7.

Figure 9.7: Internal layout of a WAL segment file



XLogLongPageHeaderData structure and XLogPageHeaderData structure are defined in src/include/access/xlog_internal.h. The explanation of both structures is omitted because those are not required in the following descriptions.

9.4 Internal layout of XLOG record

An XLOG record comprises the general header portion and each associated data portion. The first subsection describes the header structure; the remaining two subsections explain the structure of data portion in version 9.4 or earlier and in version 9.5, respectively. (The data format has changed in version 9.5.)

9.4.1 Header portion of XLOG record

All XLOG records have a general header portion defined by the structure XLogRecord. Here, the structure of 9.4 and earlier versions is shown in the following, though it is changed in version 9.5.

```
typedef struct XLogRecord
{
    uint32          xl_tot_len;    /* total len of entire record */
    TransactionId   xl_xid;        /* xact id */
    uint32          xl_len;        /* total len of rmgr data */
    uint8           xl_info;       /* flag bits, see below */
    RmgrId          xl_rmid;       /* resource manager for this record */
    /* 2 bytes of padding here, initialize to zero */
    XLogRecPtr      xl_prev;       /* ptr to previous record in log */
    pg_crc32        xl_crc;        /* CRC for this record */
} XLogRecord;
```

Apart from two variables, most of the variables would be so obvious that no need for description.

Both **xl_rmid** and **xl_info** are variables related to **resource managers**, which are collections of operations associated with the WAL feature such as writing and replaying of XLOG records. The number of resource managers tends to increase with each PostgreSQL version, Version 10 contains the following them:

Operation	Resource manager
Heap tuple operations	RM_HEAP, RM_HEAP2
Index operations	RM_BTREE, RM_HASH, RM_GIN, RM_GIST, RM_SPGIST, RM_BRIN
Sequence operations	RM_SEQ
Transaction operations	RM_XACT, RM_MULTIXACT, RM_CLOG, RM_XLOG, RM_COMMIT_TS
Tablespace operations	RM_SMGR, RM_DBASE, RM_TBLSPC, RM_RELMAP
replication and hot standby operations	RM_STANDBY, RM_REPLORIGIN, RM_GENERIC_ID, RM_LOGICALMSG_ID

Here are some representative examples how resource managers work in the following:

- If INSERT statement is issued, the header variables *xl_rmid* and *xl_info* of its XLOG record are set to *'RM_HEAP'* and *'XLOG_HEAP_INSERT'* respectively. When recovering the database cluster, the RM_HEAP's function *heap_xlog_insert()* selected according to the *xl_info* replays this XLOG record.
- Though it is similar for UPDATE statement, the header variable *xl_info* of the XLOG record is set to *'XLOG_HEAP_UPDATE'*, and the RM_HEAP's function *heap_xlog_update()* replays its record when the database recovers.

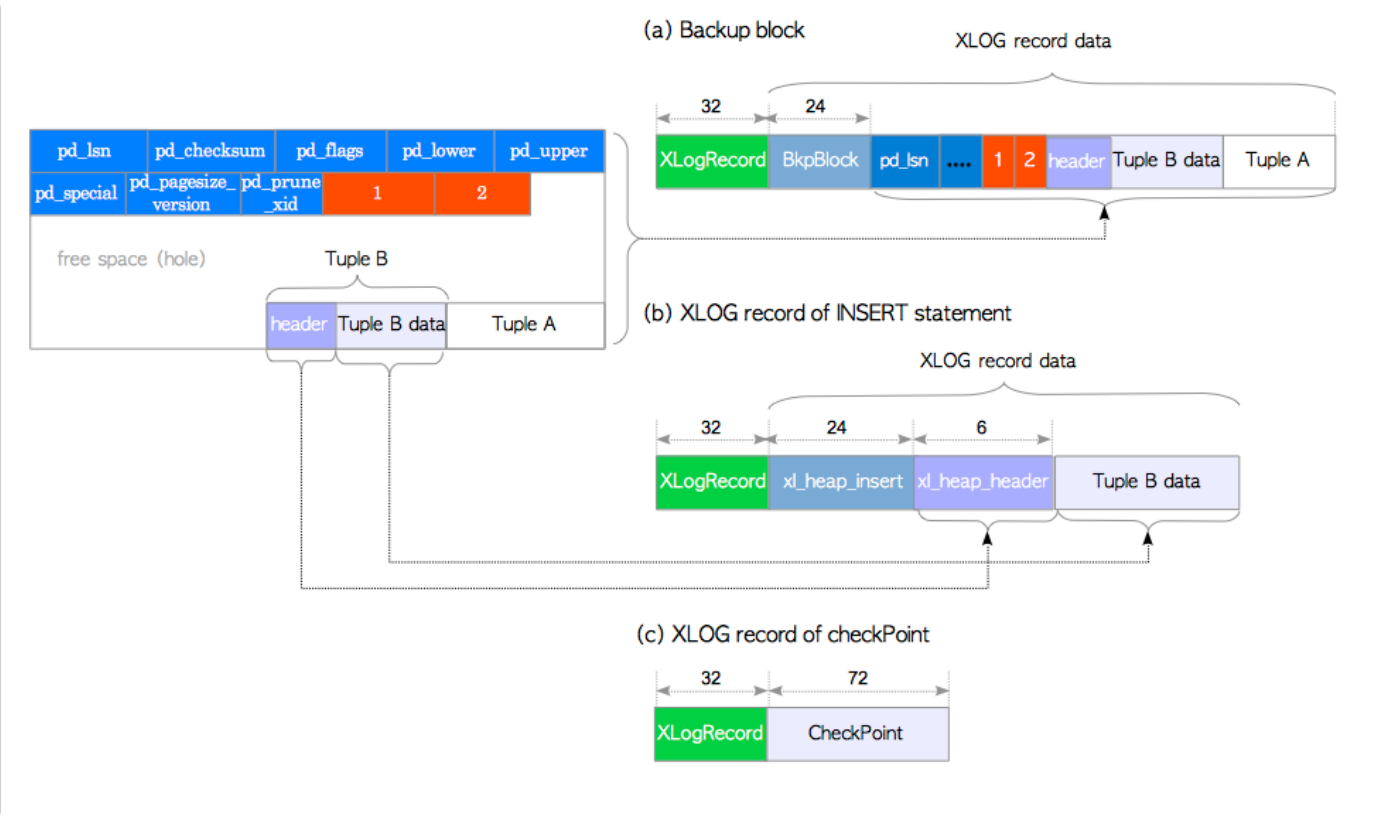
XLogRecord structure in version 9.4 or earlier is defined in src/include/access/xlog.h and that of version 9.5 or later is defined in src/include/access/xlogrecord.h.

The heap_xlog_insert and heap_xlog_update are defined in src/backend/access/heap/heapam.c; while the function xact_redo_commit is defined in src/backend/access/transam/xact.c.

9.4.2 Data portion of XLOG record (version 9.4 or earlier)

Data portion of XLOG record is classified into either backup block (entire page) or non-backup block (different data by operation).

Figure 9.8: Examples of XLOG records (version 9.4 or earlier)



The internal layouts of XLOG records are described below, using some specific examples.

Backup block

A backup block is shown in Figure 9.8 (a). It is composed of two data structures and one data object as shown below:

1. the structure XLogRecord (header-portion)
2. the structure BkpBlock
3. the entire page apart from its free-space

The BkpBlock contains the variables to identify this page in the database cluster (i.e. the *relfilenode* and the *fork number* of the relation that contains this page, and this page's *block number*), and the beginning position and the length of this page's free space.

Non-backup block

In non-backup blocks, the layout of data portion differs depending on each operation. Here, an INSERT statement's XLOG record is explained as a representative example. See Figure 9.8 (b). In this case, the XLOG record of the INSERT statement is composed of two data structures and one data object as shown below:

1. the structure XLogRecord (header-portion)
2. the structure xl_heap_insert
3. the inserted tuple – to be precise, a few bytes is removed from the tuple

The structure xl_heap_insert, contains the variables to identify the inserted tuple in the database cluster (i.e. the *relfilenode* of the table that contains this tuple, and this tuple's *tid*), and a *visibility flag* of this tuple.

The reason to remove a few bytes from inserted tuple is described in the source code comment of the structure xl_heap_header:

> We don't store the whole fixed part (HeapTupleHeaderData) of an inserted or updated tuple in WAL; we can save a few bytes by reconstructing the fields that are available elsewhere in the WAL record, or perhaps just plain needn't be reconstructed.

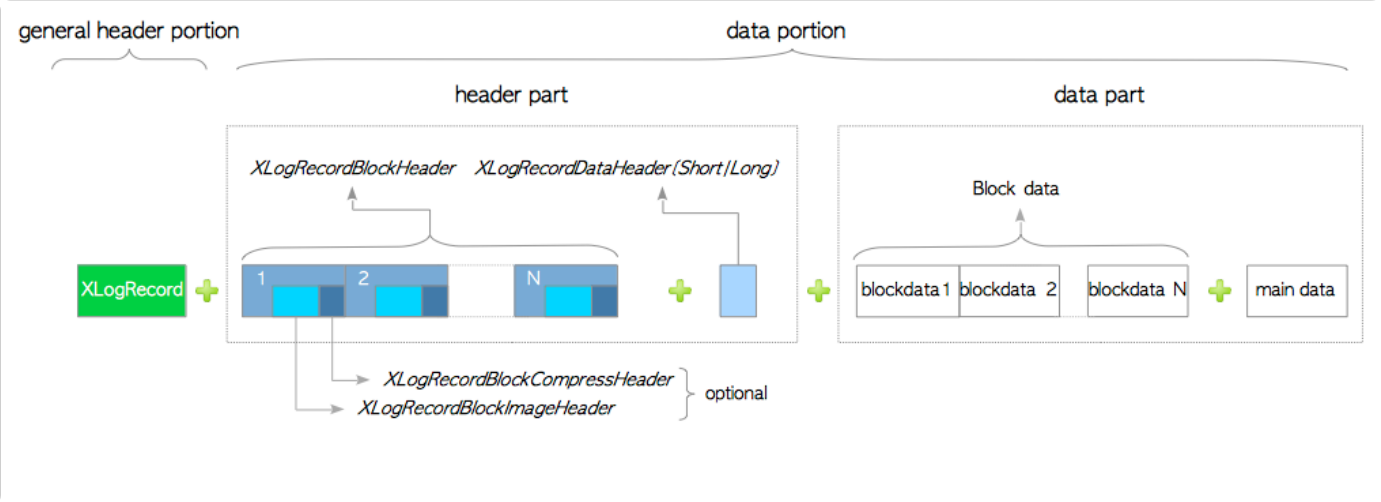
The xl_heap_header structure is defined in src/include/access/htup.h while the CheckPoint structure is defined in src/include/catalog/pg_control.h.

9.4.3 Data portion of XLOG record (version 9.5 or later)

In version 9.4 or earlier, there was no common format of XLOG record, so that each resource manager had to define one's own format. In such a case, it became increasingly difficult to maintain the source code and to implement new features related to WAL. In order to deal with this issue, a common structured format, which does not depend on resource managers, has been introduced in version 9.5.

Data portion of XLOG record can be divided into two parts: header and data. See Figure 9.9.

Figure 9.9: Common XLOG record format



Header part contains zero or more XLogRecordBlockHeaders and zero or one XLogRecordDataHeaderShort (or XLogRecordDataHeaderLong); it must contain at least either one of those. When its record stores full-page image (i.e. backup block), XLogRecordBlockHeader includes XLogRecordBlockImageHeader, and also includes XLogRecordBlockCompressHeader if its block is compressed.

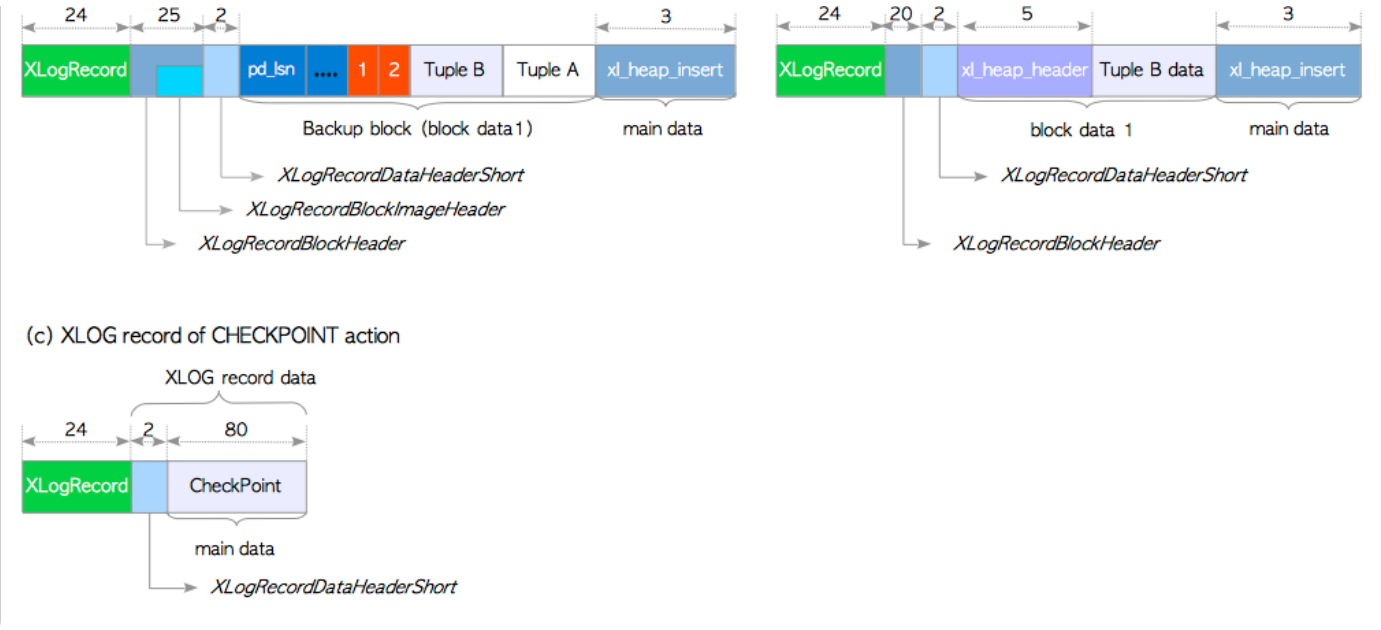
Data part is composed of zero or more block data and zero or one main data, which correspond to the XLogRecordBlockHeader(s) and to the XLogRecordDataHeader respectively.

WAL compression

In version 9.5 or later, full-page images within XLOG record can be compressed using LZ compression method by setting the parameter wal_compression = enable. In that case, the structure XLogRecordBlockCompressHeader will be added.

This feature has two advantages and one disadvantage. The advantages are reducing I/O cost for writing records and suppressing the consumption of WAL segment files. The disadvantage is consuming much CPU resource to compress.

Figure 9.10: Examples of XLOG records (version 9.5 or later)



Some specific examples are shown below as in the previous subsection.

Backup block

Backup block created by INSERT statement is shown in Figure 9.10 (a). It is composed of four data structures and one data object as shown below:

1. the structure XLogRecord (header-portion)
2. the structure XLogRecordBlockHeader including one LogRecordBlockImageHeader
3. the structure XLogRecordDataHeaderShort
4. a backup block (block data)
5. the structure xl_heap_insert (main data)

XLogRecordBlockHeader contains the variables to identify the block in the database cluster (the *relfilenode*, the *fork number*, and the *block number*); XLogRecordImageHeader contains the *length of this block* and *offset number*. (These two header structures together can store same data of BkpBlock used until version 9.4.)

XLogRecordDataHeaderShort stores the length of *xl_heap_insert* structure which is the main data of the record. (See ℹ below.)

ℹ

The main data of XLOG record which contains full-page image is not used except in some special cases (e.g. being in logical decoding and speculative insertions). It' s ignored when this record is replayed, which is the redundant data. It might be improved in the future.

In addition, main data of backup block records depend on statements which create those. For example, UPDATE statement appends *xl_heap_lock* or *xl_heap_updated*.

Non-backup block

Next, non-backup block record created by INSERT statement will be described as follows (see also Figure 9.10(b)). It is composed of four data structures and one data object as shown below:

1. the structure XLogRecord (header-portion)
2. the structure XLogRecordBlockHeader
3. the structure XLogRecordDataHeaderShort
4. an inserted tuple (to be exact, a xl_heap_header structure and an inserted data entire)
5. the structure xl_heap_insert (main data)

XLogRecordBlockHeader contains three values (the *relfilenode*, the *fork number*, and the *block number*) to specify the block inserted the tuple, and length of data portion of the inserted tuple. XLogRecordDataHeaderShort contains the length of new *xl_heap_insert* structure, which is the main data of this record.

The new *xl_heap_insert* contains only two values: *offset number* of this tuple within the block, and a *visibility flags*; it became very simple because XLogRecordBlockHeader stores most of data contained in the old one.

As the final example, a checkpoint record is shown in the Figure 9.10 (c). It is composed of three data structure as shown below:

1. the structure XLogRecord (header-portion)
2. the structure XLogRecordDataHeaderShort contained of the main data length
3. the structure CheckPoint (main data)

ℹ

The structure *xl_heap_header* is defined in src/include/access/htup.h and the CheckPoint structure is defined in src/include/catalog/pg_control.h.

Though the new format is a little complicated for us, it is well-designed for the parser of the resource managers, and also size of many types of XLOG records is usually smaller than the previous one. Size of main structures is shown in the Figure 9.8 and 9.10, so there you can calculate sizes of those records and compare each other. (The

First, issue the following statement to explore the PostgreSQL internals:

```
testdb=# INSERT INTO tbl VALUES ('A');
```

By issuing the above statement, the internal function *exec_simple_query()* is invoked; the pseudo code of *exec_simple_query()* is shown below:

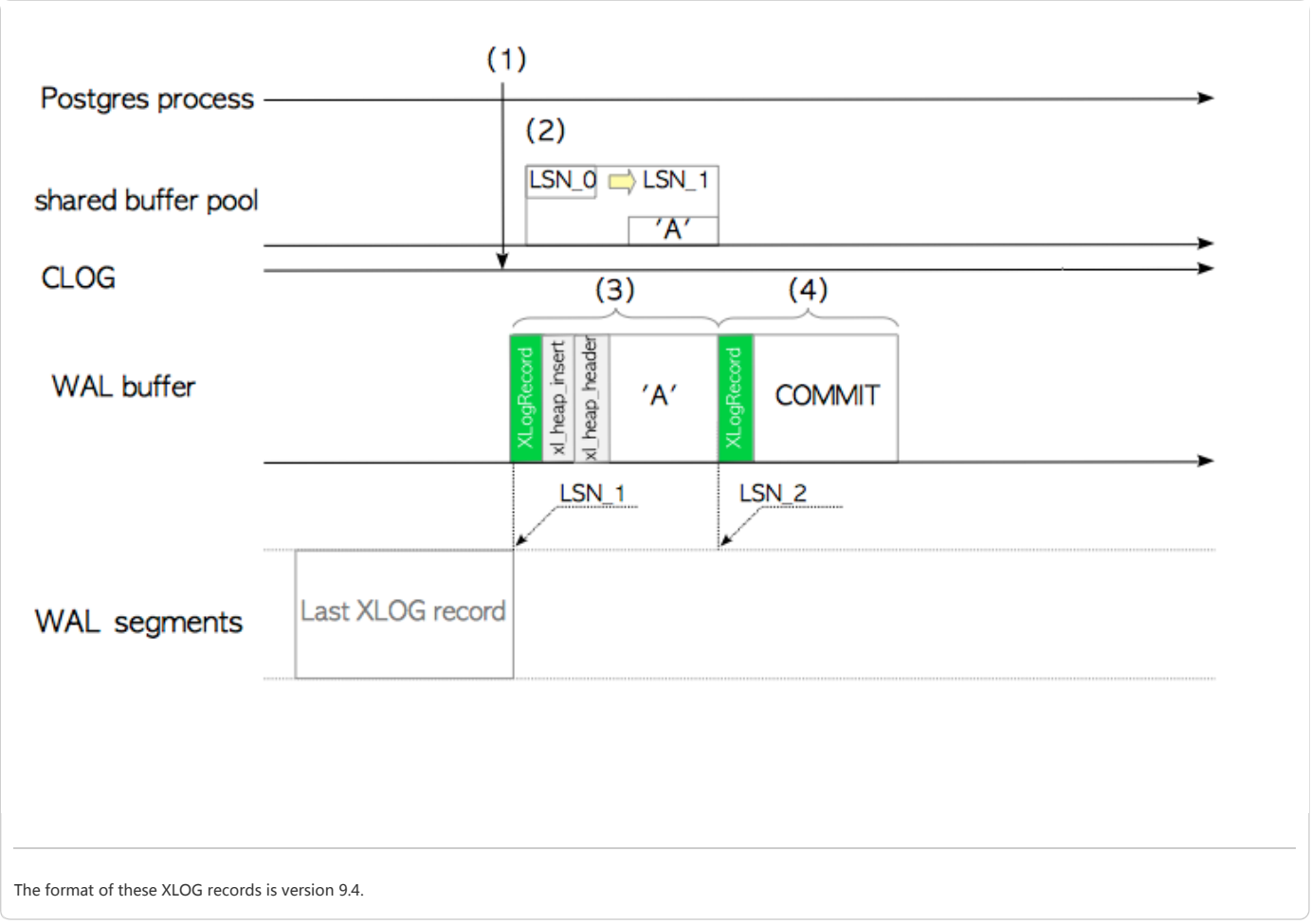
```
exec_simple_query() @postgres.c

(1) ExtendCLOG() @clog.c                /* Write the state of this transaction
                                         * "IN_PROGRESS" to the CLOG.
                                         */
(2) heap_insert()@heapam.c              /* Insert a tuple, creates a XLOG record,
                                         * and invoke the function XLogInsert.
                                         */
(3)   XLogInsert() @xlog.c (9.5 or later, xloginsert.c)
                                        /* Write the XLOG record of the inserted tuple
                                         *  to the WAL buffer, and update page's pd_lsn.
                                         */
(4) finish_xact_command() @postgres.c   /* Invoke commit action.*/
    XLogInsert() @xlog.c  (9.5 or later, xloginsert.c)
                                        /* Write a XLOG record of this commit action
                                         * to the WAL buffer.
                                         */
(5)   XLogWrite() @xlog.c               /* Write and flush all XLOG records on
                                         * the WAL buffer to WAL segment.
                                         */
(6) TransactionIdCommitTree() @transam.c  /* Change the state of this transaction
                                         * from "IN_PROGRESS" to "COMMITTED" on the CLOG.
                                         */
```

In the following paragraphs, each line of the pseudo code will be explained for understanding the writing of XLOG records; see also Figure 9.11 and 9.12.

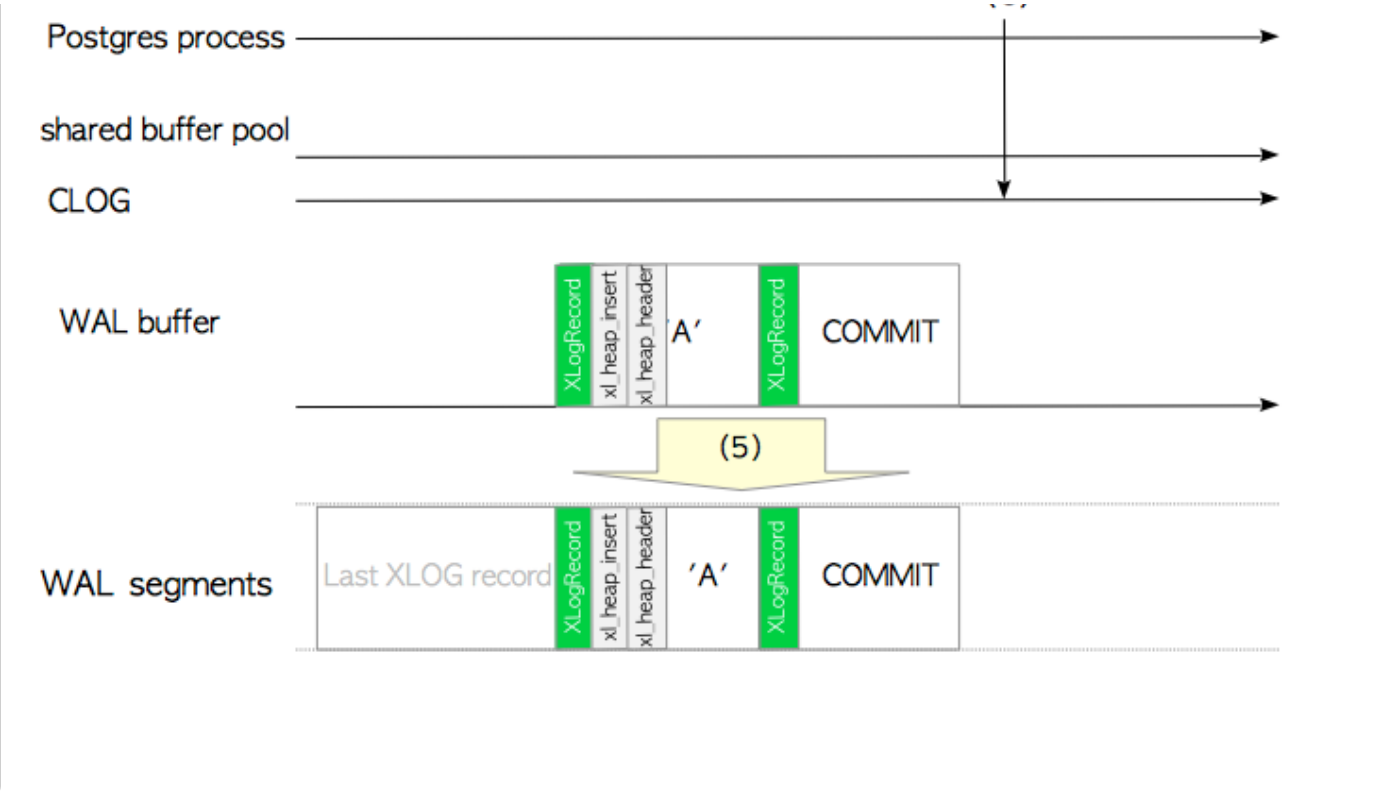
(1) The function *ExtendCLOG()* writes the state of this transaction *'IN_PROGRESS'* in the (in-memory) CLOG.
(2) The function *heap_insert()* inserts a heap tuple into the target page on the shared buffer pool, creates this page's XLOG record, and invokes the function *XLogInsert()*.
(3) The function *XLogInsert()* writes the XLOG record created by the *heap_insert()* to the WAL buffer at *LSN_1*, and then updates the modified page's pd_lsn from *LSN_0* to *LSN_1*.
(4) The function *finish_xact_command()*, which invoked to commit this transaction, creates this commit action's XLOG record, and then the function *XLogInsert()* writes this record into the WAL buffer at *LSN_2*.

Figure 9.11: Write-sequence of XLOG records



The format of these XLOG records is version 9.4.

(5) The function *XLogWrite()* writes and flushes all XLOG records on the WAL buffer to the WAL segment file.
If the parameter *wal_sync_method* is set to *'open_sync'* or *'open_datasync'*, the records are synchronously written because the function writes all records with the *open()* system call specified the flag *O_SYNC* or *O_DSYNC*. If the parameter is set to *'fsync'*, *'fsync_writethrough'* or *'fdatasync'*, the respective system call – *fsync()*, *fcntl() with F_FULLFSYNC option*, or *fdatasync()* – will be executed. In any case, all XLOG records are ensured to be written into the storage.
(6) The function *TransactionIdCommitTree()* changes the state of this transaction from *'IN_PROGRESS'* to *'COMMITTED'* on the CLOG.



In the above example, commit action has caused the writing of XLOG records into the WAL segment, but such writing may be caused when any one of the following occurs:

1. One running transaction has committed or has aborted.
2. The WAL buffer has been filled up with many tuples have been written. (The WAL buffer size is set to the parameter wal_buffers.)
3. A WAL writer process writes periodically. (See the next section.)

If one of above occurs, all WAL records on the WAL buffer are written into a WAL segment file regardless of whether their transactions have been committed or not.

It is taken for granted that DML (Data Manipulation Language) operations write XLOG records, but so do non-DML operations. As described above, a commit action writes a XLOG record that contains the id of committed transaction. Another example may be a checkpoint action to write a XLOG record that contains general information of this checkpoint. Furthermore, SELECT statement creates XLOG records in special cases, though it does not usually create them. For example, if deletion of unnecessary tuples and defragmentation of the necessary tuples in pages occur by HOT(Heap Only Tuple) during a SELECT statement processing, the XLOG records of modified pages are written into the WAL buffer.

9.6 WAL writer process

WAL writer is a background process to check the WAL buffer periodically and write all unwritten XLOG records into the WAL segments. The purpose of this process is to avoid burst of writing of XLOG records. If this process has not been enabled, the writing of XLOG records might have been bottlenecked when a large amount of data committed at one time.

WAL writer is working by default and cannot be disabled. Check interval is set to the configuration parameter *wal_writer_delay*, default value is 200 milliseconds.

9.7 Checkpoint process in PostgreSQL

In PostgreSQL, the checkpointer (background) process performs checkpoints; its process starts when one of the following occurs:

1. The interval time set for *checkpoint_timeout* from the previous checkpoint has been gone over (the default interval is 300 seconds (5 minutes)).
2. In version 9.4 or earlier, the number of WAL segment files set for *checkpoint_segments* has been consumed since the previous checkpoint (the default number is 3).
3. In version 9.5 or later, the total size of the WAL segment files in the pg_xlog (in version 10 or later, pg_wal) has exceeded the value of the parameter *max_wal_size* (the default value is 1GB (64 files)).
4. PostgreSQL server stops in *smart* or *fast* mode.

Its process also does it when a superuser issues CHECKPOINT command manually.

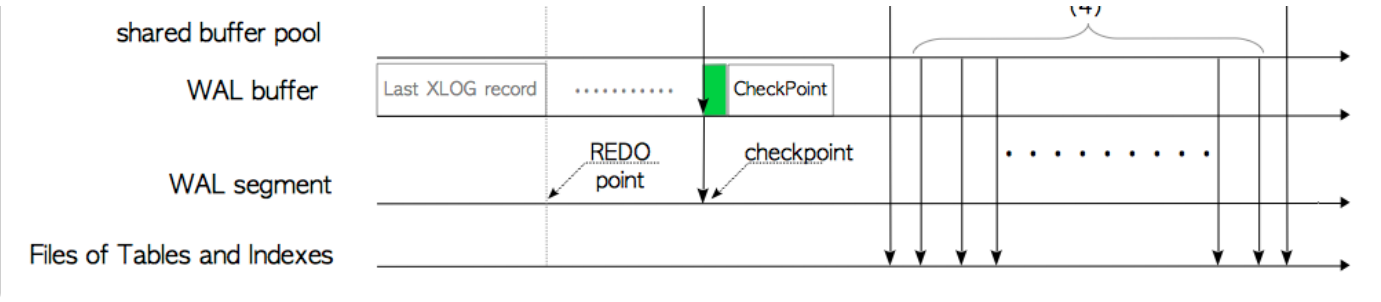
> In version 9.1 or earlier, the background writer process did both checkpinting and dirty-page writing.

In the following subsections, the outline of checkpointing and the pg_control file, which holds the metadata of the current checkpoint, are described.

9.7.1 Outline of the checkpoint processing

Checkpoint process has two aspects: the preparation of database recovery, and the cleaning of dirty pages on the shared buffer pool. In this subsection, its internal processing will be described with focusing on the former one. See Figure 9.13 and the following description.

Figure 9.13: Internal processing of PostgreSQL's checkpoint



(1) After a checkpoint process starts, the REDO point is stored in memory; REDO point is the location to write the XLOG record at the moment when the latest checkpoint is started, and is the starting point of database recovery.
(2) A XLOG record of this checkpoint (i.e. checkpoint record) is written to the WAL buffer. The data-portion of the record is defined by the structure CheckPoint, which contains several variables such as the REDO point stored with step (1).
In addition, the location to write checkpoint record is literally called the *checkpoint*.
(3) All data in shared memory (e.g. the contents of the clog, etc..) are flushed to the storage.
(4) All dirty pages on the shared buffer pool are written and flushed into the storage, gradually.
(5) The *pg_control file* is updated. This file contains the fundamental information such as the location where the checkpoint record has written (a.k.a. checkpoint location). The details of this file later.

To summarize the description above from the viewpoint of the database recovery, checkpointing creates the checkpoint record which contains the REDO point, and stores the checkpoint location and more into the *pg_control file*. Therefore, PostgreSQL enables to recover itself by replaying WAL data from the REDO point (obtained from the checkpoint record) provided by the pg_control file.

9.7.2 pg_control file

As the *pg_control file* contains the fundamental information of the checkpoint, it is certainly essential for database recovery. If it is broken or unreadable, the recovery process cannot start up in order to not obtained a starting point.

Even though *pg_control file* stores over 40 items, three items to be required in the next section are shown in the following:

- **State** – The state of database server at the time of the latest checkpointing starts. There are seven states in total: *'start up'* is the state that system is starting up; *'shut down'* is the state that system is going down normally by the shutdown command; *'in production'* is the state that system is running; and so on.
- **Latest checkpoint location** – LSN Location of the latest checkpoint record.
- **Prior checkpoint location** – LSN Location of the prior checkpoint record.

A pg_control file is stored in the global subdirectory under the base-directory; its contents can be shown using the *pg_controldata* utility.

```
postgres> pg_controldata  /usr/local/pgsql/data
pg_control version number:            937
Catalog version number:               201405111
Database system identifier:           6035535450242021944
Database cluster state:               in production
pg_control last modified:             Thu Jul 6 15:16:38 2017
Latest checkpoint location:           0/C000F48
Prior checkpoint location:            0/C000E70

... snip ...
```

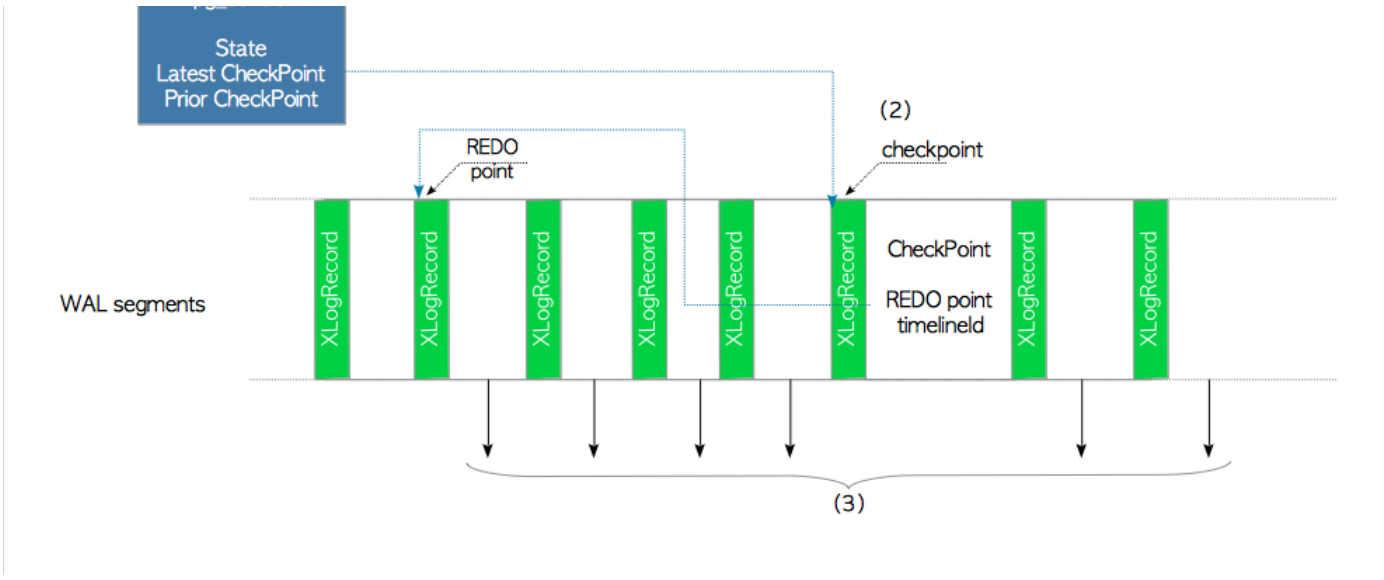
9.8 Database recovery in PostgreSQL

PostgreSQL implements the redo log based recovery feature. Should the database server crash, PostgreSQL restore the database cluster by sequentially replaying the XLOG records in the WAL segment files from the REDO point.

We had already talked about the database recovery several times up to this section, so I will describe two things regarding the recovery which has not been explained yet.

The first thing is how PostgreSQL begin the recovery process. When PostgreSQL starts up, it reads the pg_control file at first. The followings are the details of the recovery processing from that point. See Figure 9.14 and the following description.

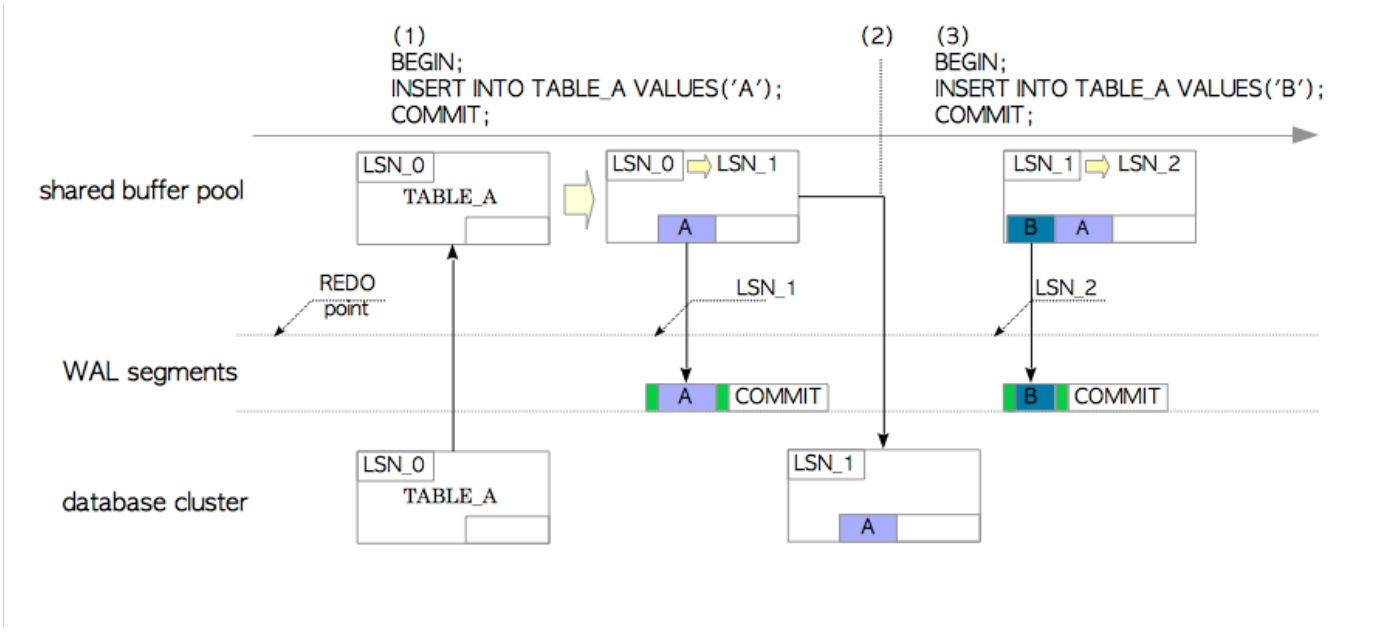
Figure 9.14: Details of the recovery process



(1) PostgreSQL reads all items of the *pg_control file* when it starts. If the *state* item is in *'in production'*, PostgreSQL will go into recovery-mode because it means that the database was not stopped normally; if *'shut down'*, it will go into normal startup-mode.
(2) PostgreSQL reads the latest checkpoint record, which location is written in the *pg_control file*, from the appropriate WAL segment file, and gets the REDO point from the record. If the latest checkpoint record is invalid, PostgreSQL reads the one prior to it. If both records are unreadable, it gives up recovering by itself.
(3) Proper resource managers read and replay XLOG records in sequence from the REDO point until they come to the last point of the latest WAL segment. When a XLOG record is replayed and if it is a backup block, it will be overwritten on the corresponding table's page regardless of its LSN. Otherwise, a (non-backup block's) XLOG record will be replayed only if the LSN of this record is larger than the *pd_lsn* of a corresponding page.

The second point is about the comparison of LSNs: why the non-backup block's LSN and the corresponding page's *pd_lsn* should be compared. Unlike the previous examples, it' s to be explained using a specific example emphasizing the need of comparison between both LSNs. See Figure 9.15 and 9.16. (Note that the WAL buffer is omitted to simplify the description.)

Figure 9.15: Insertion operations during the background writer working

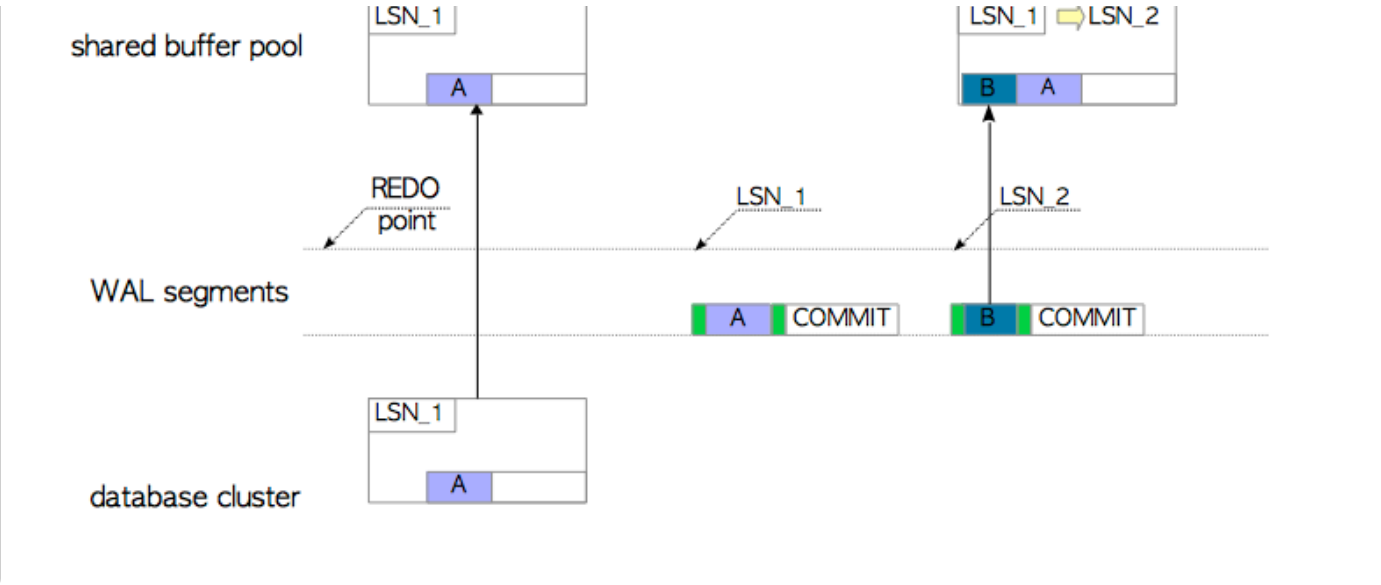


(1) PostgreSQL inserts a tuple into the TABLE_A, and writes a XLOG record at *LSN_1*.
(2) The background-writer process writes the TABLE_A's page into the storage. At this point, this page's *pd_lsn* is *LSN_1*.
(3) PostgreSQL inserts a new tuple into the TABLE_A, and writes a XLOG record at *LSN_2*. The modified page is not written into the storage yet.

Unlike the examples in overview, the TABLE_A's page has been once written into the storage in this scenario.

Do shutdown with immediate-mode, and start.

Figure 9.16: Database recovery



(1) PostgreSQL loads the first XLOG record and the TABLE_A's page, but does not replay it because this record's LSN is not larger than the TABLE_A's LSN (both values are *LSN_1*). In fact, it is clear at a glance that there is no need to replay it.
(2) Next, PostgreSQL replays the second XLOG record because this record's LSN (*LSN_2*) is larger than the current TABLE_A's LSN (*LSN_1*).

As can be seen from this example, if the replaying order of non-backup blocks is incorrect or non-backup blocks are replayed out more than once, the database cluster will no longer be consistent. In short, the redo (replay) operation of non-backup block is **not** *idempotent*. Therefore, to preserve the correct replaying order, non-backup block records should replay if and only if its LSN is greater than the corresponding page's pd_lsn.

On the other hand, as the redo operation of backup block is *idempotent*, backup blocks can be replayed any number of times regardless of its LSN.

9.9 WAL segment files Management

PostgreSQL writes XLOG records to one of the WAL segment files stored in the pg_xlog subdirectory (in version 10 or later, pg_wal subdirectory), and switches for a new one if the old one has been filled up. The number of the WAL files will vary depending on several configuration parameters, as well as server activity. In addition, their management policy has been improved in version 9.5.

In the following subsections, switching and managing of WAL segment files are described.

9.9.1 WAL segment switches

WAL segment switches occur when one of the following occurs:

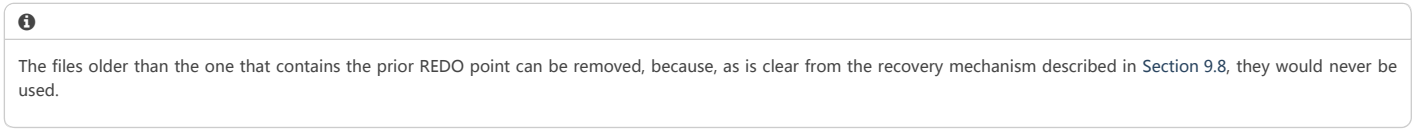
1. WAL segment has been filled up.
2. The function *pg_switch_xlog* has been issued.
3. *archive_mode* is enabled and the time set to *archive_timeout* has been exceeded.

Switched file is usually recycled (renamed and reused) for future use but it may be removed later if not necessary.

9.9.2 WAL segment management in version 9.5 or later

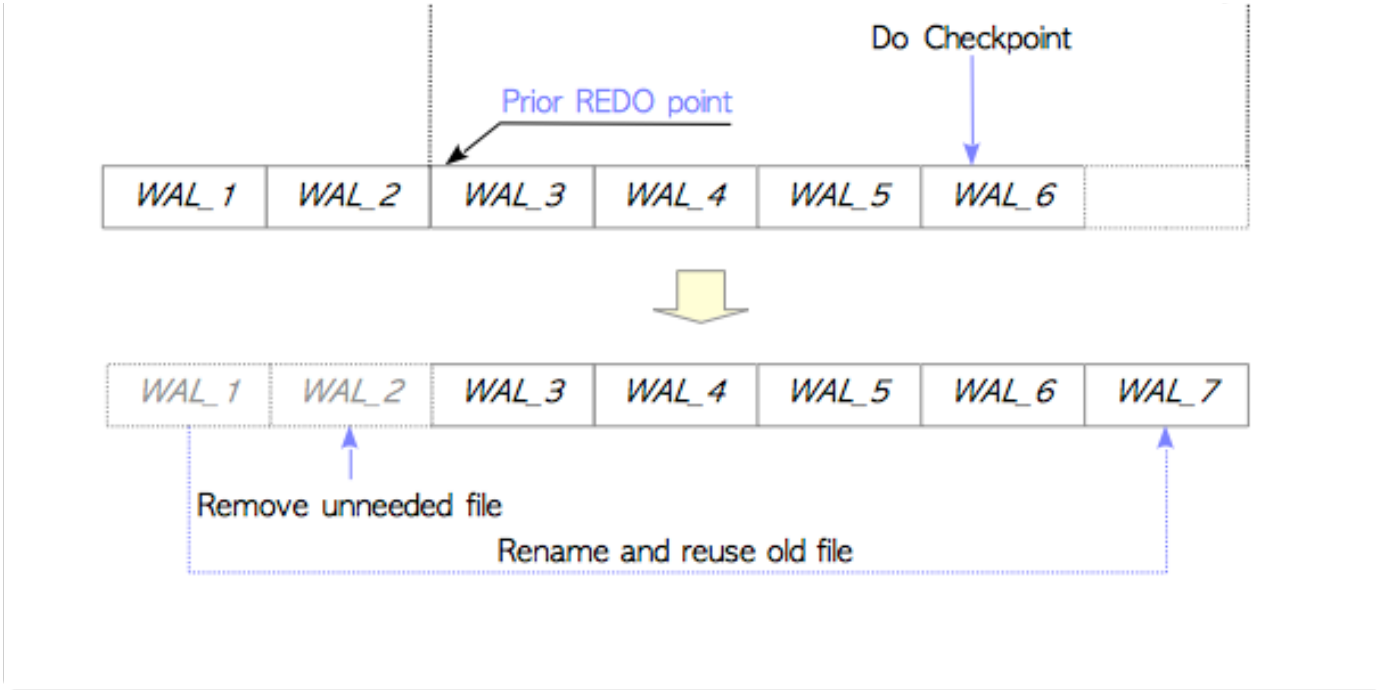
Whenever the checkpoint starts, PostgreSQL estimates and prepares the number of WAL segment files required for the next checkpoint cycle. Such estimate is made with regards to the numbers of files consumed in previous checkpoint cycles. They are counted from the segment that contains the prior REDO point, and the value is to be between *min_wal_size* (by default, 80 MB, i.e. 5 files) and *max_wal_size* (1 GB, i.e. 64 files). If a checkpoint starts, necessary files will be held or recycled, while the unnecessary ones removed.

A specific example is shown in Figure 9.17. Assuming that there are six files before checkpoint starts, *WAL_3* contains the prior REDO point, and PostgreSQL estimates that five files are needed. In this case, *WAL_1* will be renamed as *WAL_7* for recycling and *WAL_2* will be removed.



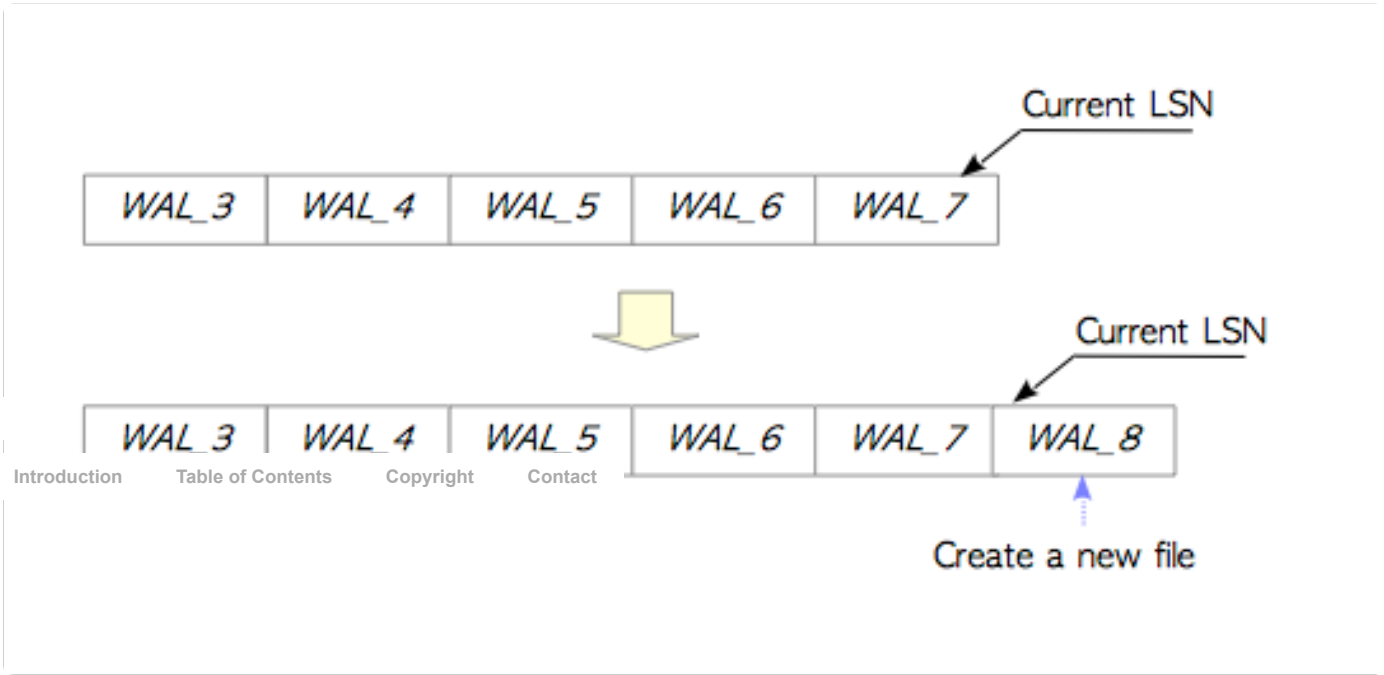
The files older than the one that contains the prior REDO point can be removed, because, as is clear from the recovery mechanism described in Section 9.8, they would never be used.

Figure 9.17: Recycling and removing WAL segment files at a checkpoint



If more files are required due to a spike in WAL activity, new files will be created while the total size of WAL files is less than *max_wal_size*. For example, in Figure 9.18, if *WAL_7* has been filled up, *WAL_8* is newly created.

Figure 9.18: Creating WAL segment file

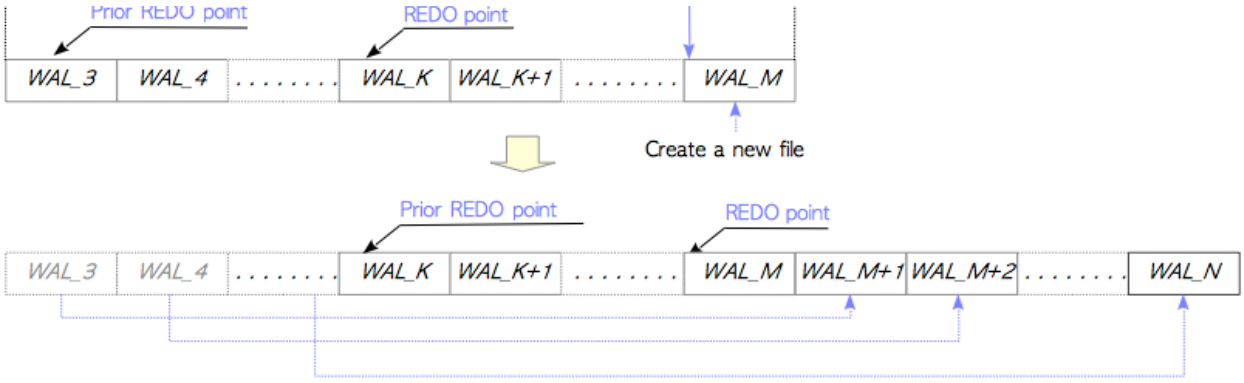


Introduction Table of Contents Copyright Contact

The number of WAL files adaptively changes depending on the server activity. If the amount of WAL data writing has constantly increased, the estimated number of the WAL segment files as well as the total size of WAL files also gradually increase. In the opposite case (i.e. the amount of WAL data writing has decreased), these decrease.

If the total size of the WAL files exceeds *max_wal_size*, a checkpoint will be started. Figure 9.19 illustrates this situation. By checkpointing, a new REDO point will be created and the last REDO point will be the prior one; and then unnecessary old files will be recycled. In this way, PostgreSQL will always hold just the WAL segment files needed for database recovery.

Figure 9.19: Checkpointing and recycling WAL segment files



The configuration parameter *wal_keep_segments* and the *replication slot* feature also affect the number of WAL segment files.

9.9.3 WAL segment management in version 9.4 or earlier

The number of WAL segment files is mainly controlled by the following three parameters: *checkpoint_segments*, *checkpoint_completion_target*, and *wal_keep_segments*. Its number is normally more than *(2 + checkpoint_completion_target) * checkpoint_segments + 1* or *checkpoint_segments + wal_keep_segments + 1* files. This number could temporarily become up to *3 * checkpoint_segments + 1* files depending on the server activity. The *replication slot* also influences the number of them.

As mentioned in Section 9.7, checkpoint process occurs when the number of *checkpoint_segments* files has been consumed. It is therefore guaranteed that two or more REDO points are always included within the WAL files because the number of the files is always greater than *2*checkpoint_segments*. The same is true if it occurs by timing out. Thus, PostgreSQL will always hold enough WAL segment files (sometimes more than necessary) required for recovery.

In version 9.4 or earlier, the parameter *checkpoint_segments* is a pain in the neck. If it is set at a small number, checkpoint occurs frequently, which causes a decrease in performance, whereas if set at a large number, the huge disk space is always required for the WAL files, some of which is not always necessary.

In version 9.5, the management policy of WAL files has improved and *checkpoint_segments* has obsoleted. Therefore, the trade-off problem described above has also been resolved.

9.10 Continuous archiving and archive logs

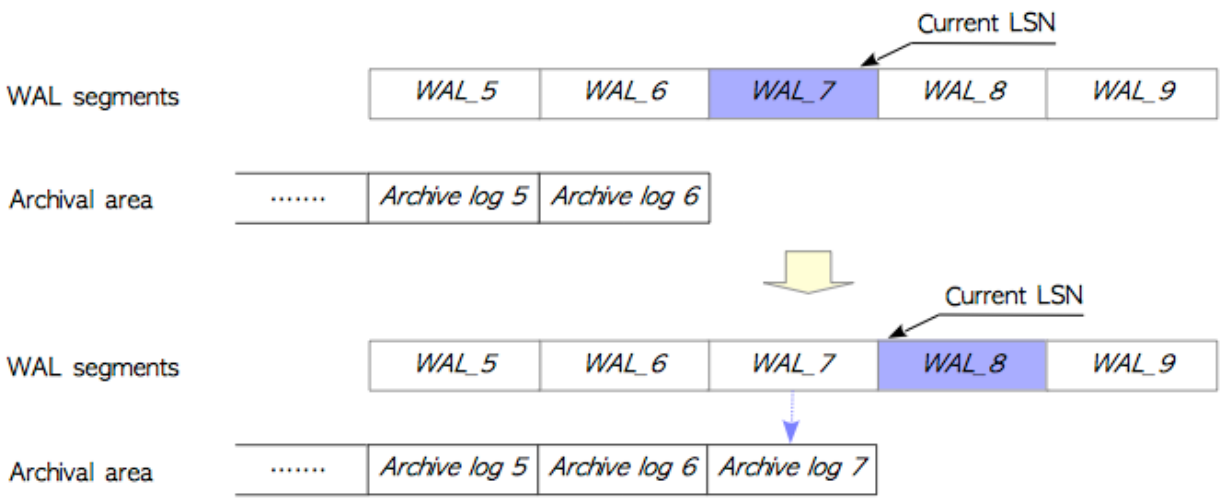
Continuous Archiving is a feature that copies WAL segment files to archival area at the time when WAL segment switches, and is performed by the *archiver (background)* process. The copied file is called an **archive log**. This feature is usually used for hot physical backup and PITR (Point-in-Time Recovery) described in Chapter 10.

The path of archival area is set to the configuration parameter *archive_command*. For example, using the following parameter, WAL segment files are copied to the directory *'/home/postgres/archives/'* every time when each segment switches:

```
archive_command = 'cp %p /home/postgres/archives/%f'
```

where, placeholder *%p* is copied WAL segment, and *%f* is archive log.

Figure 9.20: Continuous archiving



backup tools instead of ordinary copy command.

PostgreSQL does *not* clean up created archiving logs, so you should properly manage the logs when using this feature. If you do nothing, the number of archiving logs continues to increase.

The pg_archivecleanup utility is one of the useful tools for the archiving log management.