

# Chapter 1

## Database Cluster, Databases, and Tables

This chapter and the next chapter summarize the basic knowledge of PostgreSQL to help to read the subsequent chapters. This chapter describes the following topics:

- The logical structure of a database cluster
- The physical structure of a database cluster
- The internal layout of a heap table file
- The methods of writing and reading data to a table

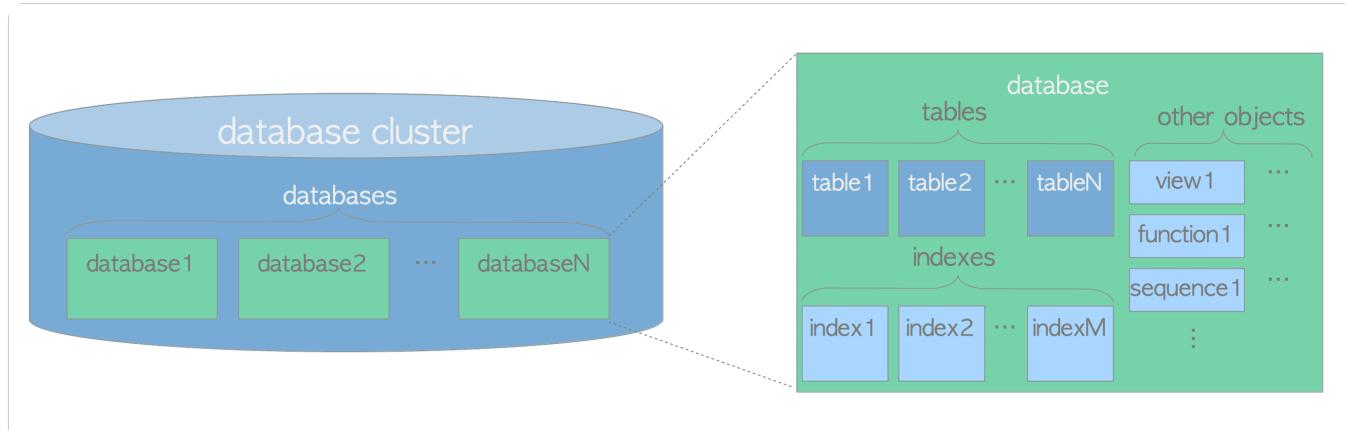
If you are already familiar with these topics, you may skip over this chapter.

### 1.1. Logical Structure of Database Cluster

A **database cluster** is a collection of *databases* managed by a PostgreSQL server. If you are hearing this definition for the first time, you might be wondering what it means. The term 'database cluster' in PostgreSQL does **not** mean 'a group of database servers'. A PostgreSQL server runs on a single host and manages a single database cluster.

Figure 1.1 shows the logical structure of a database cluster. A *database* is a collection of *database objects*. In the relational database theory, a *database object* is a data structure used to store or reference data. A (heap) *table* is a typical example, and there are many others, such as indexes, sequences, views, functions. In PostgreSQL, databases themselves are also database objects and are logically separated from each other. All other database objects (e.g., tables, indexes, etc) belong to their respective databases.

**Fig. 1.1. Logical structure of a database cluster.**



All the database objects in PostgreSQL are internally managed by respective **object identifiers (OIDs)**, which are unsigned 4-byte integers. The relations between database objects and their respective OIDs are stored in appropriate system catalogs, depending on the type of objects. For example, OIDs of databases and heap tables are stored in `pg_database` and `pg_class` respectively. You can find out the OIDs you want to know by issuing the queries such as the following:

```
sampledb=# SELECT datname, oid FROM pg_database WHERE datname = 'sampledb';
  datname |   oid
-----+-----
 sampledb | 16384
(1 row)

sampledb=# SELECT relname, oid FROM pg_class WHERE relname = 'sampltbl';
  relname |   oid
-----+-----
 sampltbl | 18740
(1 row)
```

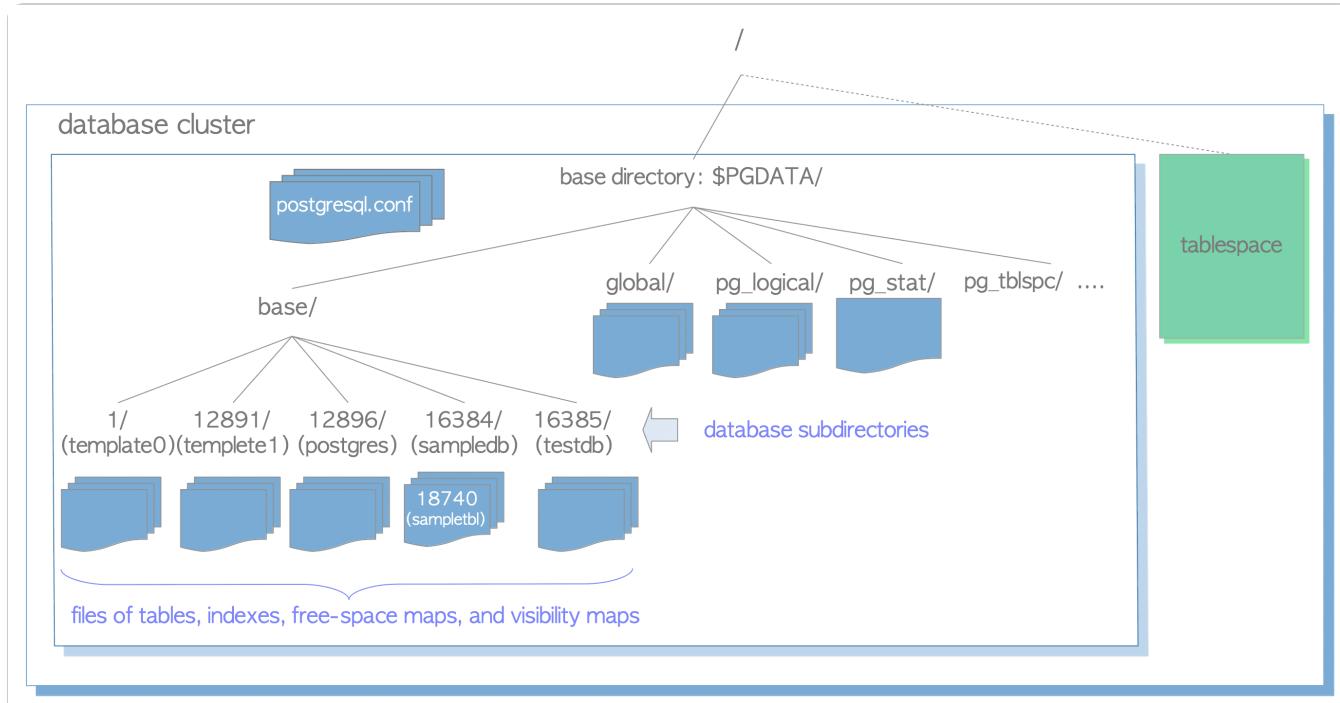
## 1.2. Physical Structure of Database Cluster

A *database cluster* is basically a single directory, referred to as **base directory**. It contains some subdirectories and many files. When you execute the `initdb` utility to initialize a new database cluster, a base directory will be created under the specified directory. The path of the base directory is usually set to the environment variable `PGDATA`.

Figure 1.2 shows an example of database cluster in PostgreSQL. A database is a subdirectory under the *base* subdirectory, and each of the tables and indexes is (at least) one file stored under the subdirectory of the database to which it belongs. There are several subdirectories containing particular data and configuration files.

While PostgreSQL supports *tablespaces*, the meaning of the term is different from other RDBMSs. A tablespace in PostgreSQL is a single directory that contains some data outside of the base directory.

**Fig. 1.2. An example of database cluster.**



In the following subsections, the layout of a database cluster, databases, files associated with tables and indexes, and tablespaces in PostgreSQL are described.

### 1.2.1. Layout of a Database Cluster

The layout of database cluster has been described in the official document. Main files and subdirectories in a part of the document have been listed in Table 1.1:

**table 1.1: Layout of files and subdirectories under the base directory (From the official document)**

files	description
PG_VERSION	A file containing the major version number of PostgreSQL.
pg_hba.conf	A file to control PostgreSQL's client authentication.
pg_ident.conf	A file to control PostgreSQL's user name mapping.
postgresql.conf	A file to set configuration parameters.
postgresql.auto.conf	A file used for storing configuration parameters that are set in ALTER SYSTEM. (versions 9.4 or later)
postmaster.opts	A file recording the command line options the server was last started with.
subdirectories	description
base/	Subdirectory containing per-database subdirectories.
global/	Subdirectory containing cluster-wide tables, such as pg_database and pg_control.
pg_commit_ts/	Subdirectory containing transaction commit timestamp data. (versions 9.5 or later)
pg_clog/ (versions 9.6 or earlier)	Subdirectory containing transaction commit state data. It is renamed to pg_xact in version 10. CLOG will be described in Section 5.4.
pg_dynshmem/	Subdirectory containing files used by the dynamic shared memory subsystem. (versions 9.4 or later)
pg_logical/	Subdirectory containing status data for logical decoding. (versions 9.4 or later)
pg_multixact/	Subdirectory containing multitransaction status data. (used for shared row locks)
pg_notify/	Subdirectory containing LISTEN/NOTIFY status data.
pg_repslot/	Subdirectory containing replication slot data. (versions 9.4 or later)
pg_serial/	Subdirectory containing information about committed serializable transactions. (versions 9.1 or later)
pg_snapshots/	Subdirectory containing exported snapshots. The PostgreSQL's function pg_export_snapshot creates a snapshot information file in this subdirectory. (versions 9.2 or later)
pg_stat/	Subdirectory containing permanent files for the statistics subsystem.
pg_stat_tmp/	Subdirectory containing temporary files for the statistics subsystem.
pg_subtrans/	Subdirectory containing subtransaction status data.

<code>pg_tblspc/</code>	Subdirectory containing symbolic links to tablespaces.
<code>pg_twophase/</code>	Subdirectory containing state files for prepared transactions.
<code>pg_wal/</code> (versions 10 or later)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed from <code>pg_xlog</code> in Version 10.
<code>pg_xact/</code> (versions 10 or later)	Subdirectory containing transaction commit state data. It is renamed from <code>pg_clog</code> in Version 10. CLOG will be described in Section 5.4.
<code>pg_xlog/</code> (versions 9.6 or earlier)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed to <code>pg_wal</code> in Version 10.

## 1.2.2. Layout of Databases

A database is a subdirectory under the `base` subdirectory. The database directory names are identical to the respective OIDs. For example, when the OID of the database `sampledb` is 16384, its subdirectory name is 16384.

```
$ cd $PGDATA
$ ls -ld base/16384
drwx----- 213 postgres postgres 7242 8 26 16:33 16384
```

## 1.2.3. Layout of Files Associated with Tables and Indexes

Each table or index whose size is less than 1GB is stored in a single file under the database directory to which it belongs. Tables and indexes are internally managed by individual OIDs, while their data files are managed by the variable, `reldfilenode`. The `reldfilenode` values of tables and indexes basically but **not** always match the respective OIDs, the details are described below.

For example, let's show the OID and `reldfilenode` of the table `sampletbl`:

```
sampledb=# SELECT relname, oid, reldfilenode FROM pg_class WHERE relname = 'sampletbl';
   relname |   oid   | reldfilenode
-----+-----+
 sampletbl | 18740 |          18740
(1 row)
```

As you can see, the `oid` and `reldfilenode` values are equal in this case. You can also see that the data file path of the table `sampletbl` is '`base/16384/18740`'.

```
$ cd $PGDATA
$ ls -la base/16384/18740
-rw----- 1 postgres postgres 8192 Apr 21 10:21 base/16384/18740
```

The `reldfilenode` values of tables and indexes can be changed by issuing certain commands, such as TRUNCATE, REINDEX, CLUSTER. For example, if we truncate the table `sampletbl`, PostgreSQL will assign a new `reldfilenode` (18812) to the table, removes the old data file (18740), and creates a new one (18812).

```
sampledb=# TRUNCATE sampletbl;
TRUNCATE TABLE

sampledb=# SELECT relname, oid, reldfilenode FROM pg_class WHERE relname = 'sampletbl';
   relname |   oid   | reldfilenode
-----+-----+
 sampletbl | 18740 |          18812
(1 row)
```



In versions 9.0 or later, the built-in function `pg_relation_filepath` is useful as this function returns the file path name of the relation with the specified OID or name.

```
sampledb=# SELECT pg_relation_filepath('sampletbl');
 pg_relation_filepath
-----
 base/16384/18812
(1 row)
```

When the file size of tables and indexes exceeds 1GB, PostgreSQL creates a new file named like `reldatalocation.1` and uses it. If the new file is filled up, PostgreSQL will create another new file named like `reldatalocation.2` and so on.

```
$ cd $PGDATA
$ ls -la -h base/16384/19427*
-rw----- 1 postgres postgres 1.0G Apr 21 11:16 data/base/16384/19427
-rw----- 1 postgres postgres 45M Apr 21 11:20 data/base/16384/19427.1
...
```



The maximum file size of tables and indexes can be changed using the configuration option `--with-segsize` when building PostgreSQL.

If you look carefully at the database subdirectories, you will find that each table has two associated files, suffixed with `_fsm` and `_vm`. Those are called the **free space map** and **visibility map**, respectively.

The free space map stores information about the free space capacity on each page within the table file, and the visibility map stores information about the visibility of each page within the table file. (More details can be found in Sections 5.3.4 and 6.2.)

Indexes only have individual free space maps and do not have visibility map.

A specific example is shown below:

```
$ cd $PGDATA
$ ls -la base/16384/18751*
-rw----- 1 postgres postgres 8192 Apr 21 10:21 base/16384/18751
-rw----- 1 postgres postgres 24576 Apr 21 10:18 base/16384/18751_fsm
-rw----- 1 postgres postgres 8192 Apr 21 10:18 base/16384/18751_vm
```

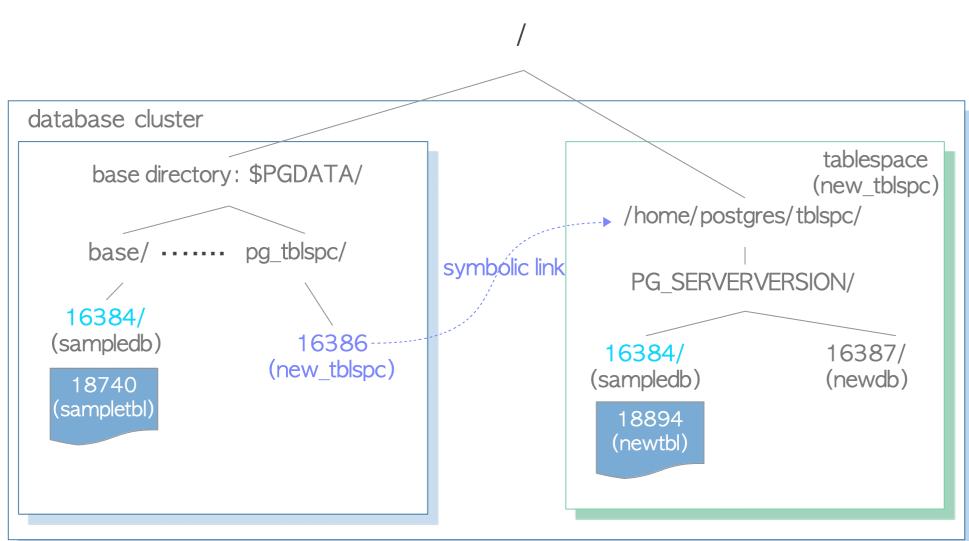
The free space map and visibility map may also be internally referred to as the **forks** of each relation. the free space map is the first fork of the table/index data file (the fork number is 1), the visibility map the second fork of the table's data file (the fork number is 2). The fork number of the data file is 0.

## 1.2.4. Tablespaces

A *tablespace* in PostgreSQL is an additional data area outside the base directory. This functionality was implemented in version 8.0.

Figure 1.3 shows the internal layout of a tablespace and its relationship with the main data area.

**Fig. 1.3. A Tablespace in the Database Cluster.**



A tablespace is created under the directory that is specified when you issue the `CREATE TABLESPACE` statement. Under that directory, a version-specific subdirectory (e.g., PG\_14\_202011044) will be created. The naming convention for the version-specific subdirectory is shown below.

PG \_ 'Major version' \_ 'Catalogue version number'

For example, if you create a tablespace 'new\_tblspc' at '/home/postgres/tblspc', with an OID of 16386, a subdirectory named 'PG\_14\_202011044' will be created under the tablespace.

```
$ ls -l /home/postgres/tblspc/
total 4
drwx----- 2 postgres postgres 4096 Apr 21 10:08 PG_14_202011044
```

The tablespace directory is addressed by a symbolic link from the `pg_tblspc` subdirectory. The link name is the same as the OID value of tablespace.

```
$ ls -l $PGDATA/pg_tblspc/
total 0
lrwxrwxrwx 1 postgres postgres 21 Apr 21 10:08 16386 -> /home/postgres/tblspc
```

If you create a new database (OID 16387) under the tablespace, its directory is created under the version-specific subdirectory.

```
$ ls -l /home/postgres/tblspc/PG_14_202011044/
total 4
drwx----- 2 postgres postgres 4096 Apr 21 10:10 16387
```

If you create a new table that belongs to the database created under the base directory, first, a new directory is created under the version-specific subdirectory. The name of the new directory is the same as the OID of the existing database. Then, the new table file is placed under the created directory.

```

sampledb=# CREATE TABLE newtbl (.....) TABLESPACE new_tblspc;
sampledb=# SELECT pg_relation_filepath('newtbl');
      pg_relation_filepath
-----
pg_tblspc/16386/PG_14_202011044/16384/18894

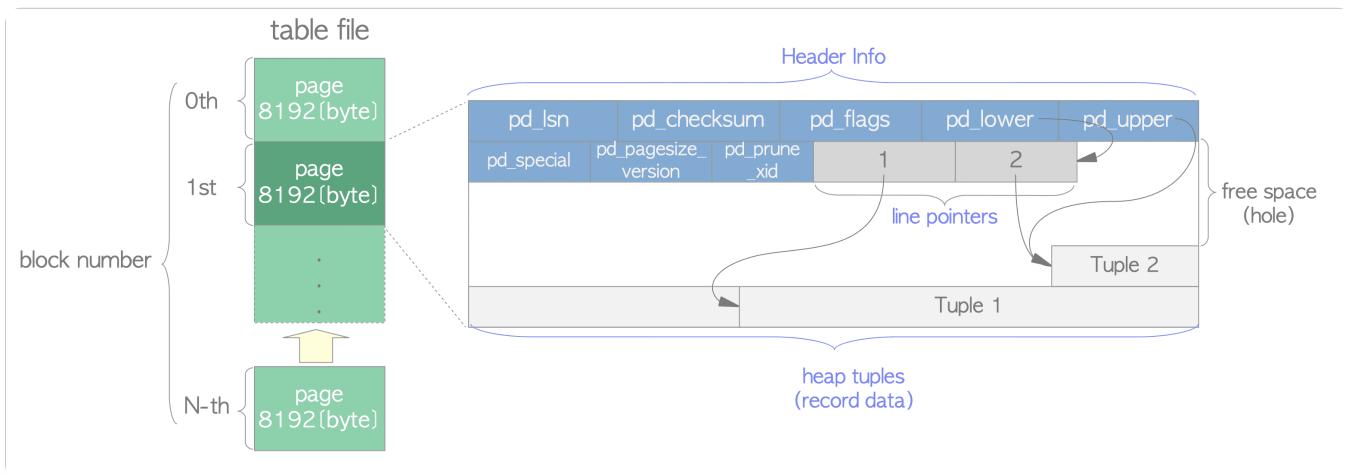
```

## 1.3. Internal Layout of a Heap Table File

Inside a data file (heap table, index, free space map, and visibility map), it is divided into **pages** (or **blocks**) of fixed length, which is 8192 bytes (8 KB) by default. The pages within each file are numbered sequentially from 0, and these numbers are called **block numbers**. If the file is full, PostgreSQL adds a new empty page to the end of the file to increase the file size.

The internal layout of pages depends on the data file type. In this section, the table layout is described, as this information will be required in the following chapters.

**Fig. 1.4. Page layout of a heap table file.**



A page within a table contains three kinds of data:

1. **heap tuple(s)** – A heap tuple is a record data itself. Heap tuples are stacked in order from the bottom of the page. The internal structure of tuple is described in Section 5.2 and Chapter 9, as it requires knowledge of both concurrency control (CC) and write-ahead logging (WAL) in PostgreSQL.

2. **line pointer(s)** – A line pointer is 4 bytes long and holds a pointer to each heap tuple. It is also called an **item pointer**.

Line pointers form a simple array that plays the role of an index to the tuples. Each index is numbered sequentially from 1, and called **offset number**. When a new tuple is added to the page, a new line pointer is also pushed onto the array to point to the new tuple.

3. **header data** – A header data defined by the structure `PageHeaderData` is allocated in the beginning of the page. It is 24 byte long and contains general information about the page. The major variables of the structure are described below.

- *pd\_lsn* – This variable stores the LSN of XLOG record written by the last change of this page. It is an 8-byte unsigned integer, and is related to the WAL (Write-Ahead Logging) mechanism. The details are described in Chapter 9.
- *pd\_checksum* – This variable stores the checksum value of this page. (Note that this variable is supported in versions 9.3 or later; in earlier versions, this part had stored the timelineId of the page.)

- *pd\_lower, pd\_upper* – *pd\_lower* points to the end of line pointers, and *pd\_upper* to the beginning of the newest heap tuple.
- *pd\_special* – This variable is for indexes. In the page within tables, it points to the end of the page. (In the page within indexes, it points to the beginning of special space, which is the data area held only by indexes and contains the particular data according to the kind of index types such as B-tree, GiST, GiN, etc.)

An empty space between the end of line pointers and the beginning of the newest tuple is referred to as **free space** or **hole**.

To identify a tuple within the table, a **tuple identifier (TID)** is used internally. A TID comprises a pair of values: the *block number* of the page that contains the tuple, and the *offset number* of the line pointer that points to the tuple. A typical example of its usage is `index`. See more detail in Section 1.4.2.



The structure `PageHeaderData` is defined in `src/include/storage/bufpage.h`.



In the field of computer science, this type of page is called a **slotted page**, and the line pointers correspond to a **slot array**.

In addition, heap tuple whose size is greater than about 2 KB (about 1/4 of 8 KB) is stored and managed using a method called **TOAST** (The Oversized-Attribute Storage Technique). Refer PostgreSQL documentation for details.

## 1.4. The Methods of Writing and Reading Tuples

In the end of this chapter, the methods of writing and reading heap tuples are described.

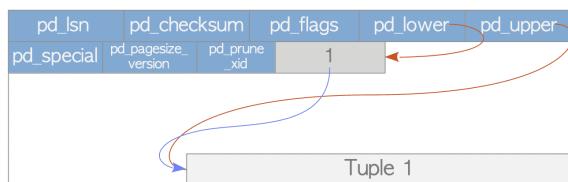
### 1.4.1. Writing Heap Tuples

Suppose a table composed of one page that contains just one heap tuple. The *pd\_lower* of this page points to the first line pointer, and both the line pointer and the *pd\_upper* point to the first heap tuple. See Fig. 1.5(a).

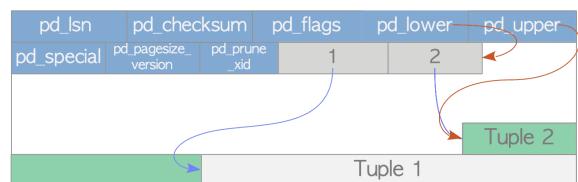
When the second tuple is inserted, it is placed after the first one. The second line pointer is appended to the first one, and it points to the second tuple. The *pd\_lower* changes to point to the second line pointer, and the *pd\_upper* to the second heap tuple. See Fig. 1.5(b). Other header data within this page (e.g., *pd\_lsn*, *pg\_checksum*, *pg\_flag*) are also updated to appropriate values; more details are described in Section 5.3 and Chapter 9.

**Fig. 1.5. Writing of a heap tuple.**

(a) Before insertion of Tuple 2



(b) After insertion of Tuple 2



## 1.4.2. Reading Heap Tuples

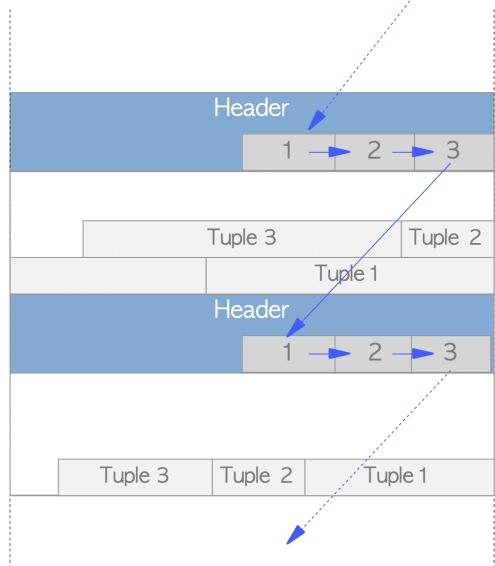
Two typical access methods, sequential scan and B-tree index scan, are outlined here:

- **Sequential scan** – It reads all tuples in all pages sequentially by scanning all line pointers in each page. See Fig. 1.6(a).
- **B-tree index scan** – It reads an index file that contains index tuples, each of which is composed of an index key and a TID that points to the target heap tuple. If the index tuple with the key that you are looking for has been found, PostgreSQL reads the desired heap tuple using the obtained TID value. (The description of how to find the index tuples in a B-tree index is not explained here, as it is very common and the space here is limited. See the relevant materials.) For example, in Fig. 1.6(b), the TID value of the obtained index tuple is '(block = 7, Offset = 2)' This means that the target heap tuple is the 2nd tuple in the 7th page within the table, so PostgreSQL can read the desired heap tuple without unnecessary scanning in the pages.

**Fig. 1.6. Sequential scan and index scan.**

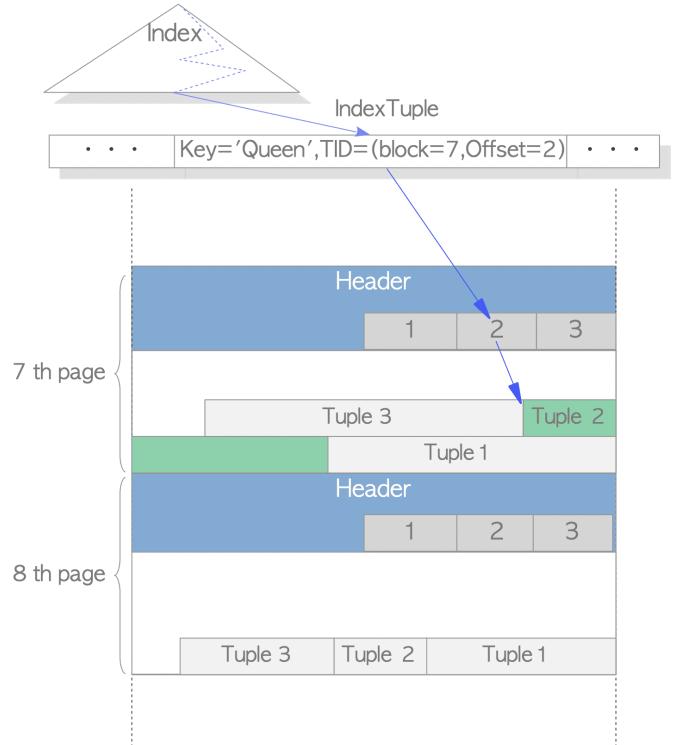
(a) Sequential Scan

SELECT \* FROM tbl;



(b) Index Scan

SELECT \* FROM tbl WHERE col = 'Queen';



This document does not explain indexes in details. To understand them, I recommend to read the valuable posts shown below:

- Indexes in PostgreSQL — 1
- Indexes in PostgreSQL — 2
- Indexes in PostgreSQL — 3 (Hash)
- Indexes in PostgreSQL — 4 (Btree)
- Indexes in PostgreSQL — 5 (GiST)
- Indexes in PostgreSQL — 6 (SP-GiST)
- Indexes in PostgreSQL — 7 (GIN)
- Indexes in PostgreSQL — 9 (BRIN)



PostgreSQL also supports TID-Scan, Bitmap-Scan, and Index-Only-Scan.

TID-Scan is a method that accesses a tuple directly by using TID of the desired tuple. For example, to find the 1st tuple in the 0-th page within the table, issue the following query:

```
sampledb=# SELECT ctid, data FROM sampletbl WHERE ctid = '(0, 1)' ;
 ctid | data
-----+
 (0, 1) | AAAAAAAA
(1 row)
```

Index-Only-Scan will be described in details in Chapter 7.

# Chapter 2

## Process and Memory Architecture

In this chapter, the process architecture and memory architecture in PostgreSQL are summarized to help to read the subsequent chapters. If you are already familiar with them, you may skip over this chapter.

### 2.1. Process Architecture

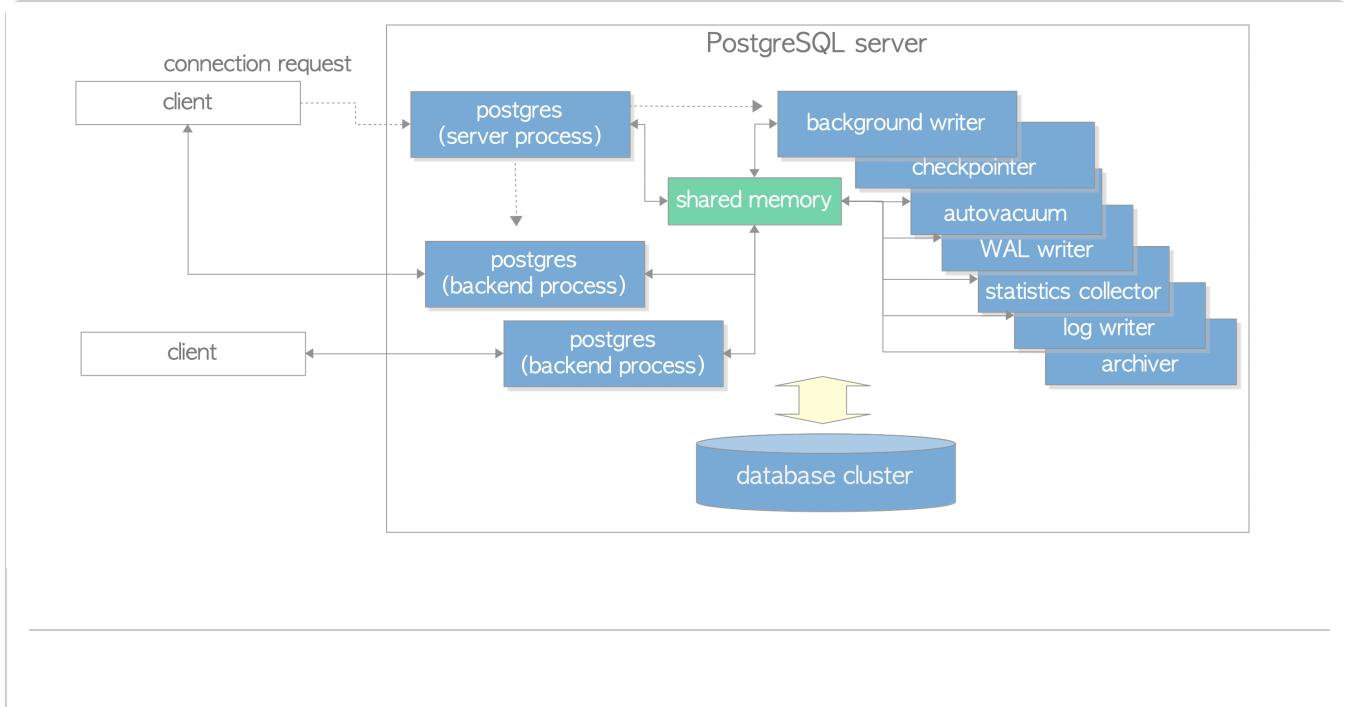
PostgreSQL is a client/server type relational database management system with a multi-process architecture that runs on a single host.

A collection of multiple processes that cooperatively manage a database cluster is usually referred to as a '*PostgreSQL server*'. It contains the following types of processes:

- The **postgres server process** is the parent of all processes related to database cluster management.
- Each **backend process** handles all queries and statements issued by a connected client.
- Various **background processes** perform tasks for database management, such as VACUUM and CHECKPOINT processing.
- **Replication-associated processes** perform streaming replication. More details are described in Chapter 11.
- **Background worker processes** supported from version 9.3, it can perform any processing implemented by users. For more information, refer to the official document.

In the following subsections describe the details of the first three types of processes.

**Fig. 2.1. An example of the process architecture in PostgreSQL.**



This figure shows processes of a PostgreSQL server: a *postgres* server process, two backend processes, seven background processes, and two client processes. The database cluster, the shared memory, and two client processes are also illustrated.

## 2.1.1. Postgres Server Process

As already described above, a *postgres server process* is a parent of all processes in a PostgreSQL server. In the earlier versions, it was called 'postmaster'.

When you execute the `pg_ctl` utility with `start` option, a *postgres* server process starts up. It then allocates a shared memory area in memory, starts various background processes, starts replication-associated processes and background worker processes if necessary, and waits for connection requests from clients. Whenever it receives a connection request from a client, it starts a backend process. (The started backend process then handles all queries issued by the connected client.)

A *postgres* server process listens to one network port, the default port is 5432. Although more than one PostgreSQL server can be run on the same host, each server must be set to listen to different port number, such as 5432, 5433, and so on.

## 2.1.2. Backend Processes

A *backend process*, also called a *postgres* process, is started by the *postgres* server process and handles all queries issued by one connected client. It communicates with the client using a single TCP connection and terminates when the client disconnects.

Since a *backend process* is only allowed to operate on one database, you must explicitly specify the database you want to use when connecting to a PostgreSQL server.

PostgreSQL allows multiple clients to connect simultaneously. the configuration parameter `max_connections` controls the maximum number of the clients (default is 100).

If many clients, such as WEB applications, frequently connect and disconnect from a PostgreSQL server, it can increase the cost of establishing connections and creating *backend processes*, as PostgreSQL does not have a native connection pooling feature. This can have a negative impact on the performance of the database server. To deal with such a case, a connection pooling middleware such as `pgbouncer` or `pgpool-II`) is usually used.

## 2.1.3. Background Processes

Table 2.1 shows a list of background processes. In contrast to the *postgres* server process and the *backend process*, it is impossible to explain each of the functions simply. This is because these functions depend on the individual specific features and PostgreSQL internals. Therefore, in this chapter, only introductions are made. Details will be described in the following chapters.

**Table 2.1: background processes.**

process	description	reference
background writer	This process writes dirty pages on the shared buffer pool to a persistent storage (e.g., HDD, SSD) on a regular basis gradually.	Section 8.6

	(In versions 9.1 or earlier, it was also responsible for the checkpoint process.)	
checkpointer	This process performs the checkpoint process in versions 9.2 or later.	Section 8.6, Section 9.7
autovacuum launcher	This process periodically invokes the autovacuum-worker processes for the vacuum process. (More precisely, it requests the postgres server to create the autovacuum workers.)	Section 6.5
WAL writer	This process writes and flushes the WAL data on the WAL buffer to persistent storage periodically.	Section 9.9
statistics collector	This process collects statistics information such as for pg_stat_activity and pg_stat_database, etc.	
logging collector (logger)	This process writes error messages into log files.	
archiver	This process executes archiving logging.	Section 9.10



The actual processes of a PostgreSQL server are shown below. In the following example, there is one postgres server process (pid is 9687), two backend processes (pids are 9697 and 9717), and several background processes listed in Table 2.1. See also Figure 2.1.

```
postgres> pstree -p 9687
-+= 00001 root /sbin/launchd
\-- 09687 postgres /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
|--- 09688 postgres postgres: logger process
|--- 09690 postgres postgres: checkpointer process
|--- 09691 postgres postgres: writer process
|--- 09692 postgres postgres: wal writer process
|--- 09693 postgres postgres: autovacuum launcher process
|--- 09694 postgres postgres: archiver process
|--- 09695 postgres postgres: stats collector process
|--- 09697 postgres postgres: postgres sampledb 192.168.1.100(54924) idle
\--- 09717 postgres postgres: postgres sampledb 192.168.1.100(54964) idle in transaction
```

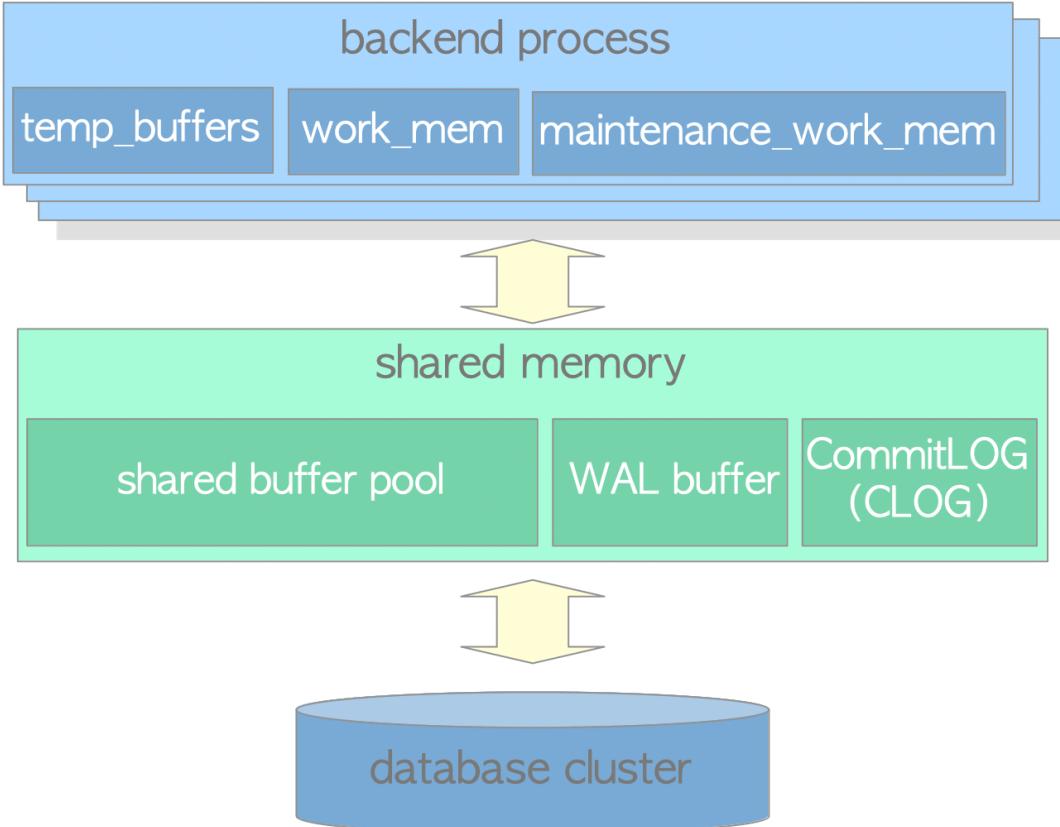
## 2.2. Memory Architecture

Memory architecture in PostgreSQL can be classified into two broad categories:

- Local memory area – allocated by each backend process for its own use.
- Shared memory area – used by all processes of a PostgreSQL server.

In the following subsections, those are briefly described.

**Fig. 2.2. Memory architecture in PostgreSQL.**



## 2.2.1. Local Memory Area

Each backend process allocates a local memory area for query processing. The area is divided into several sub-areas, whose sizes are either fixed or variable. Table 2.2 shows a list of the major sub-areas. The details of each sub-area will be described in the following chapters.

**Table 2.2: Local memory area**

sub-area	description	reference
<code>work_mem</code>	The executor uses this area for sorting tuples by ORDER BY and DISTINCT operations, and for joining tables by merge-join and hash-join operations.	Chapter 3
<code>maintenance_work_mem</code>	Some kinds of maintenance operations (e.g., VACUUM, REINDEX) use this area.	Section 6.1
<code>temp_buffers</code>	The executor uses this area for storing temporary tables.	

## 2.2.2. Shared Memory Area

A shared memory area is allocated by a PostgreSQL server when it starts up. This area is also divided into several fixed-sized sub-areas. Table 2.3 shows a list of the major sub-areas. The details will be described in the following chapters.

**Table 2.3: Shared memory area**

sub-area	description	reference
shared buffer pool	PostgreSQL loads pages within tables and indexes from a persistent storage to this area, and operates them directly.	Chapter 8
WAL buffer	To ensure that no data has been lost by server failures, PostgreSQL supports the WAL mechanism. WAL data (also referred to as XLOG records) are the transaction log in PostgreSQL. The WAL buffer is a buffering area of the WAL data before writing to a persistent storage.	Chapter 9
commit log	The commit log (CLOG) keeps the states of all transactions (e.g., in_progress, committed, aborted) for the concurrency control (CC) mechanism.	Section 5.4

In addition to the shared buffer pool, WAL buffer, and commit log, PostgreSQL allocates several other areas, as shown below:

- Sub-areas for the various access control mechanisms. (e.g., semaphores, lightweight locks, shared and exclusive locks, etc)
- Sub-areas for the various background processes, such as the checkpointer and autovacuum.
- Sub-areas for transaction processing, such as savepoints and two-phase commit.

and others.

# Chapter 3

## Query Processing (Part 1)

---

**A**s described in the official document, PostgreSQL supports a very large number of features required by the SQL standard of 2011. Query processing is the most complicated subsystem in PostgreSQL, and it efficiently processes the supported SQL. This chapter outlines this query processing, in particular, it focuses on query optimization.

This chapter comprises the following three parts:

**Part 1:** Section 3.1.

This section provides an overview of query processing in PostgreSQL.

**Part 2:** Sections 3.2. — 3.4.

This part explains the steps followed to obtain the optimal plan of a single-table query. In Sections 3.2 and 3.3, the processes of estimating the cost and creating the plan tree are explained, respectively. Section 3.4 briefly describes the operation of the executor.

**Part 3:** Sections 3.5. — 3.6.

This part explains the process of obtaining the optimal plan of a multiple-table query. In Section 3.5, three join methods are described: nested loop, merge, and hash join. Section 3.6 explains the process of creating the plan tree of a multiple-table query.

PostgreSQL supports three technically interesting and practical features: Foreign Data Wrappers (FDW), Parallel Query and JIT compilation which is supported from version 11. The first two of them will be described in Chapter 4. The JIT compilation is out of scope of this document; see the official document in details.

[Go to Part 2, Part 3 ➔](#)

### 3.1. Overview

In PostgreSQL, although the parallel query implemented in version 9.6 uses multiple background worker processes, a backend process basically handles all queries issued by the connected client. This backend consists of five subsystems:

1. Parser

The parser generates a parse tree from an SQL statement in plain text.

2. Analyzer/Analyser

The analyzer/analyser carries out a semantic analysis of a parse tree and generates a query tree.

3. Rewriter

The rewriter transforms a query tree using the rules stored in the rule system if such rules exist.

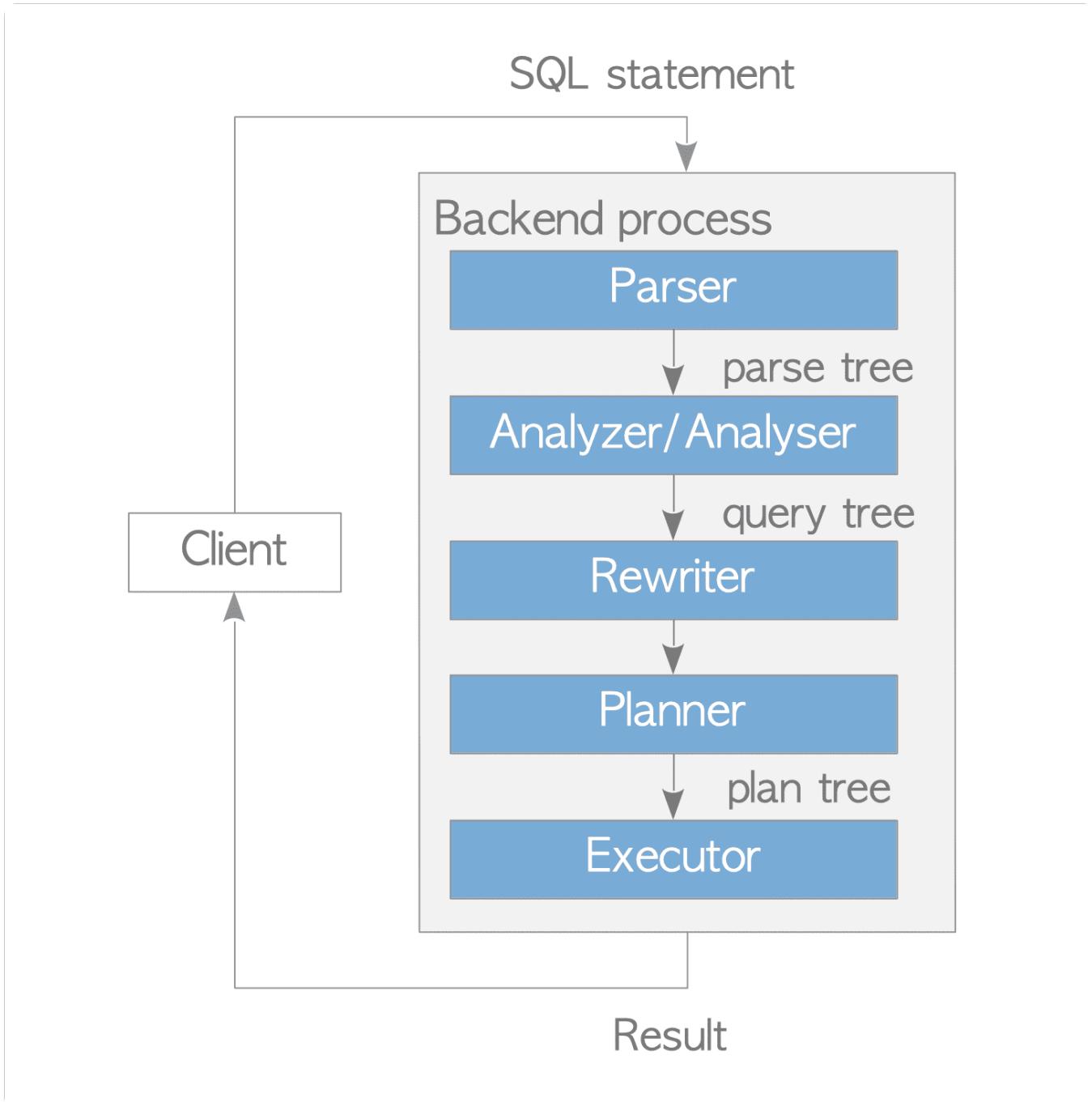
#### 4. Planner

The planner generates the plan tree that can most effectively be executed from the query tree.

#### 5. Executor

The executor executes the query by accessing the tables and indexes in the order that was created by the plan tree.

**Fig. 3.1. Query Processing.**



In this section, an overview of these subsystems is provided. Due to the fact that the planner and the executor are very complicated, a detailed explanation for these functions will be provided in the following sections.



PostgreSQL's query processing is described in the official document in detail.

### 3.1.1. Parser

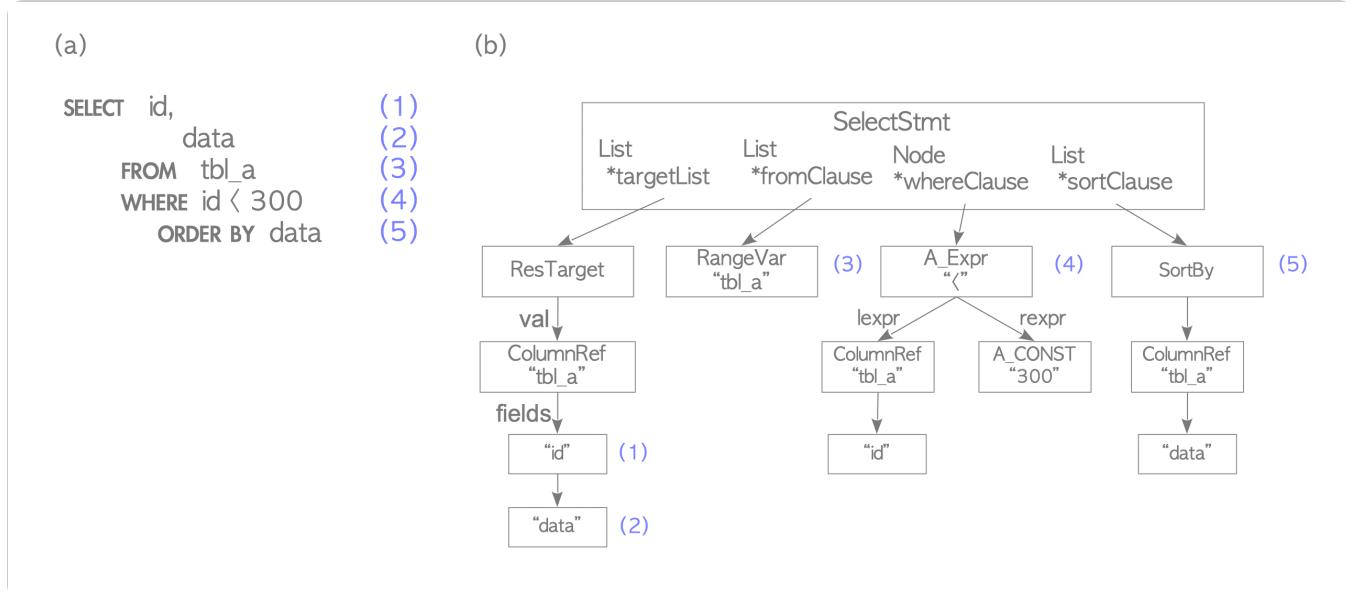
The parser generates a parse tree that can be read by subsequent subsystems from an SQL statement in plain text. Here is a specific example, without a detailed description.

Let us consider the query shown below.

```
testdb=# SELECT id, data FROM tbl_a WHERE id < 300 ORDER BY data;
```

A parse tree is a tree whose root node is the `SelectStmt` structure defined in `parsenodes.h`. Figure 3.2(b) illustrates the parse tree of the query shown in Fig. 3.2(a).

**Fig. 3.2. An example of a parse tree.**



The elements of the `SELECT` query and the corresponding elements of the parse tree are numbered the same. For example, (1) is an item of the first target list, and it is the column '`id`' of the table; (4) is a `WHERE` clause; and so on.

The parser only checks the syntax of an input when generating a parse tree. Therefore, it only returns an error if there is a syntax error in the query.

The parser does not check the semantics of an input query. For example, even if the query contains a table name that does not exist, the parser does not return an error. Semantic checks are done by the analyzer/analyser.

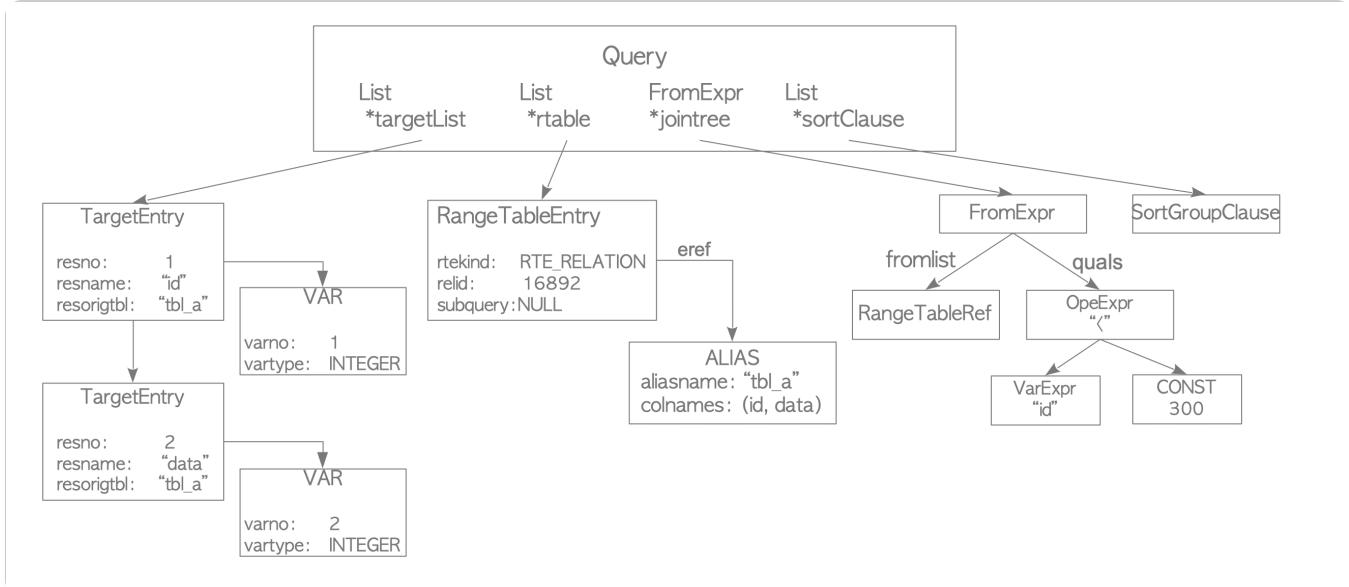
### 3.1.2. Analyzer/Analyser

The analyzer/analyser runs a semantic analysis of a parse tree generated by the parser and generates a query tree.

The root of a query tree is the `Query` structure defined in `parsenodes.h`. This structure contains metadata of its corresponding query, such as the type of the command (`SELECT`, `INSERT`, or others), and several leaves. Each leaf forms a list or a tree and holds data for the individual particular clause.

Figure 3.3 illustrates the query tree of the query shown in Fig. 3.2(a) in the previous subsection.

**Fig. 3.3. An example of a query tree.**



The above query tree is briefly described as follows:

- The targetlist is a list of columns that are the result of this query. In this example, the list is composed of two columns: '*id*' and '*data*'. If the input query tree uses '\*' (asterisk), the analyzer/analyser will explicitly replace it with all of the columns.
- The range table is a list of relations that are used in this query. In this example, the list holds the information of the table '*tbl\_a*', such as the *OID* of the table and the name of the table.
- The join tree stores the FROM clause and the WHERE clauses.
- The sort clause is a list of SortGroupClause.

The details of the query tree are described in the official document.

### 3.1.3. Rewriter

The rewriter is the system that realizes the rule system. It transforms a query tree according to the rules stored in the `pg_rules` system catalog, if necessary. The rule system is an interesting system in itself, but the descriptions of the rule system and the rewriter have been omitted to prevent this chapter from becoming too long.

#### View

Views in PostgreSQL are implemented by using the rule system. When a view is defined by the `CREATE VIEW` command, the corresponding rule is automatically generated and stored in the catalog.

Assume that the following view is already defined and the corresponding rule is stored in the `pg_rules` system catalog:

```

sampledb=# CREATE VIEW employees_list
sampledb-#   AS SELECT e.id, e.name, d.name AS department
sampledb-#     FROM employees AS e, departments AS d WHERE e.department_id = d.id;
  
```

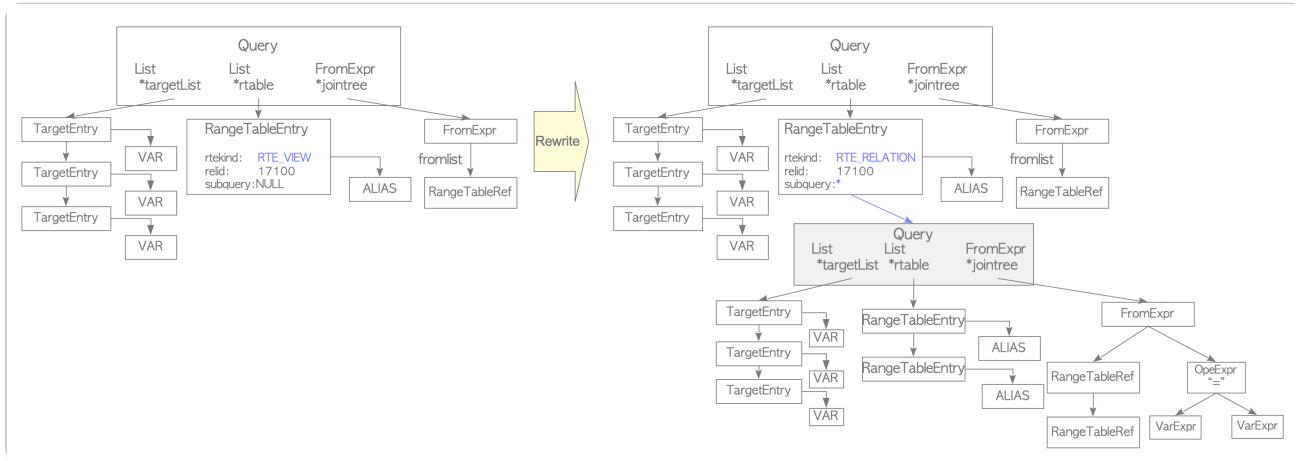
When a query that contains a view shown below is issued, the parser creates the parse tree as shown in Fig. 3.4(a).

```

sampledb=# SELECT * FROM employees_list;
  
```

At this stage, the rewriter processes the range table node to a parse tree of the subquery, which is the corresponding view, stored in *pg\_rules*.

**Fig. 3.4. An example of the rewriter stage.**



Since PostgreSQL realizes views using such a mechanism, views could not be updated until version 9.2. However, views can be updated from version 9.3 onwards; nonetheless, there are many limitations in updating the view. These details are described in the official document.

### 3.1.4. Planner and Executor

The planner receives a query tree from the rewriter and generates a (query) plan tree that can be processed by the executor most effectively.

The planner in PostgreSQL is based on pure cost-based optimization. It does not support rule-based optimization or hints. This planner is the most complex subsystem in PostgreSQL. Therefore, an overview of the planner will be provided in the subsequent sections of this chapter.

#### ● pg\_hint\_plan

PostgreSQL does not support planner hints in SQL, and it will not be supported forever. If you want to use hints in your queries, the extension referred to *pg\_hint\_plan* will be worth considering. Refer to the official site in detail.

As in other RDBMS, the EXPLAIN command in PostgreSQL displays the plan tree itself. A specific example is shown below:

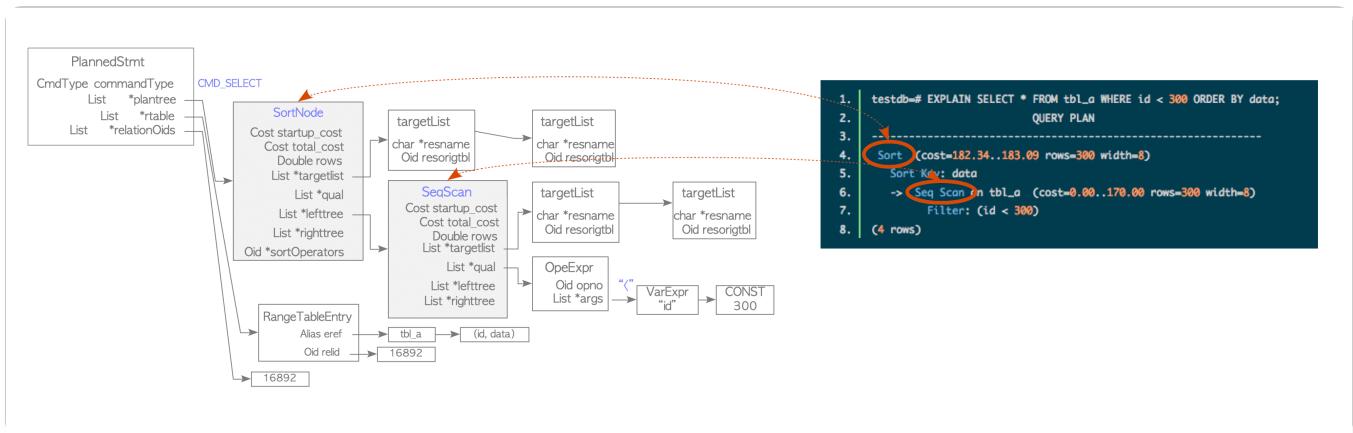
```

1. testdb=# EXPLAIN SELECT * FROM tbl_a WHERE id < 300 ORDER BY data;
2.                                     QUERY PLAN
3.
4. Sort  (cost=182.34..183.09 rows=300 width=8)
5.   Sort Key: data
6.   -> Seq Scan on tbl_a  (cost=0.00..170.00 rows=300 width=8)
7.     Filter: (id < 300)
8.   (4 rows)

```

This result shows the plan tree shown in Fig. 3.5.

**Fig. 3.5. A simple plan tree and the relationship between the plan tree and the result of the EXPLAIN command.**



A plan tree is composed of elements called *plan nodes*, and it is connected to the plantree list of the *PlannedStmt* structure. These elements are defined in *plannodes.h*. Details will be explained in Section 3.3.3 (and Section 3.5.4.2).

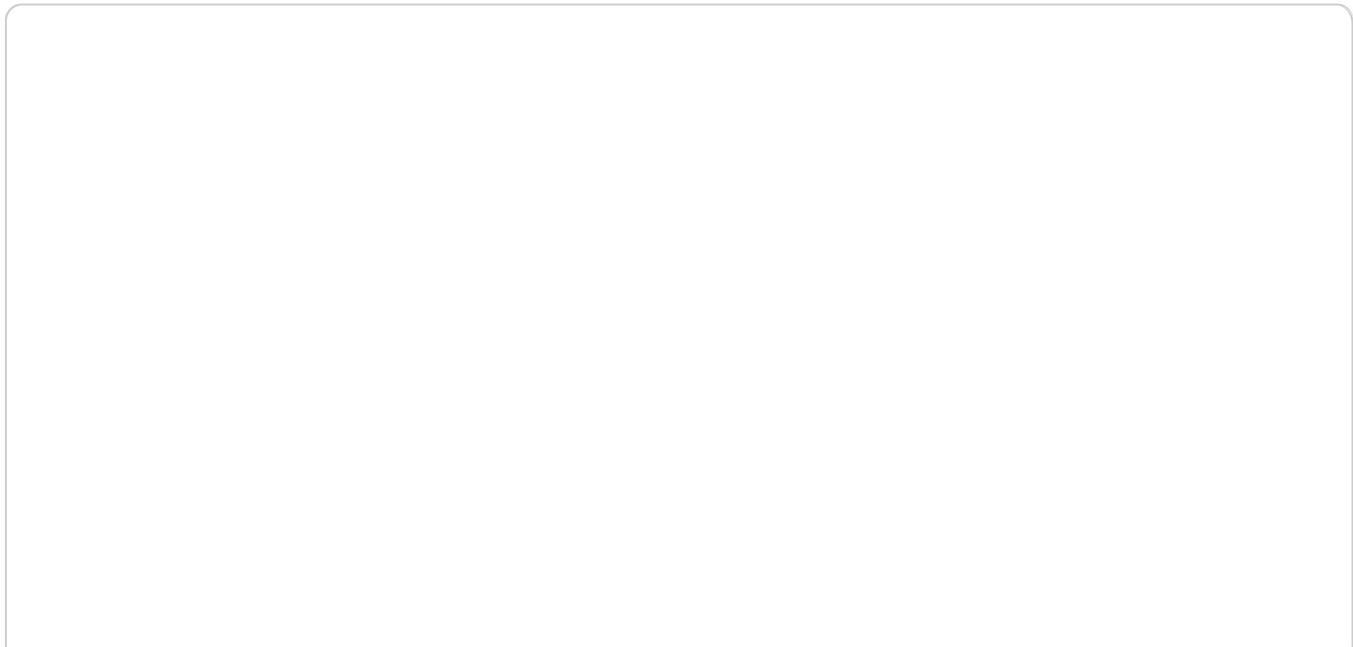
Each plan node has information that the executor requires for processing. In the case of a single-table query, the executor processes from the end of the plan tree to the root.

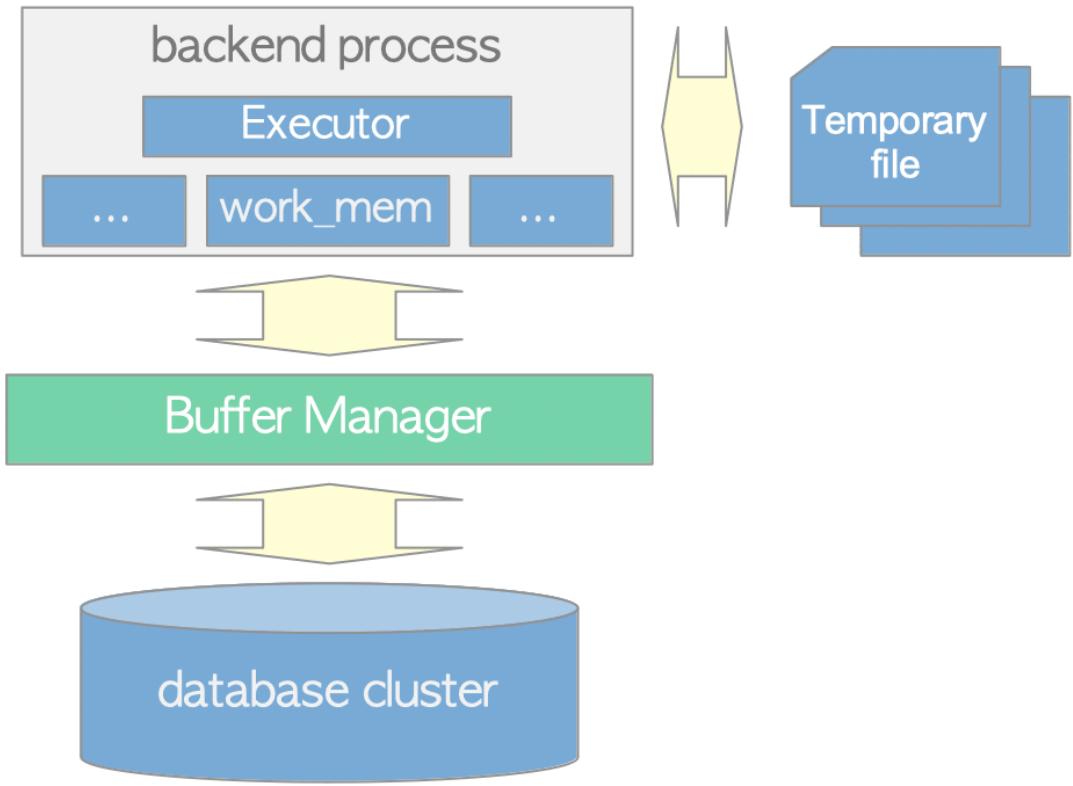
For example, the plan tree shown in Fig. 3.5 is a list of a sort node and a sequential scan node. Therefore, the executor scans the table *tbl\_a* by a sequential scan and then sorts the obtained result.

The executor reads and writes tables and indexes in the database cluster via the buffer manager described in Chapter 8. When processing a query, the executor uses some memory areas, such as *temp\_buffers* and *work\_mem*, allocated in advance and creates temporary files if necessary.

In addition, when accessing tuples, PostgreSQL uses the concurrency control mechanism to maintain consistency and isolation of the running transactions. The concurrency control mechanism is described in Chapter 5.

**Fig. 3.6. The relationship among the executor, buffer manager and temporary files.**





## 3.2. Cost Estimation in Single-Table Query

Go to Section 3.2.

Go to Part 2, Part 3 ➔

# Chapter 3

## Query Processing (Part 2)

◀ Back to Part 1      Go to Part 3 ▶

### 3.2. Cost Estimation in Single-Table Query

PostgreSQL's query optimization is based on cost. Costs are dimensionless values, and they are not absolute performance indicators, but rather indicators to compare the relative performance of operations.

Costs are estimated by the functions defined in `costsize.c`. All operations executed by the executor have corresponding cost functions. For example, the costs of sequential scans and index scans are estimated by `cost_seqscan()` and `cost_index()`, respectively.

In PostgreSQL, there are three kinds of costs: **start-up**, **run** and **total**. The total cost is the sum of the start-up and run costs, so only the start-up and run costs are independently estimated.

- The **start-up** cost is the cost expended before the first tuple is fetched. For example, the start-up cost of the index scan node is the cost of reading index pages to access the first tuple in the target table.
- The **run** cost is the cost of fetching all tuples.
- The **total** cost is the sum of the costs of both start-up and run costs.

The `EXPLAIN` command shows both of start-up and total costs in each operation. The simplest example is shown below:

```
1. testdb=# EXPLAIN SELECT * FROM tbl;
2.                                     QUERY PLAN
3. -----
4. Seq Scan on tbl  (cost=0.00..145.00 rows=10000 width=8)
5. (1 row)
```

In Line 4, the command shows information about the sequential scan. In the cost section, there are two values; 0.00 and 145.00. In this case, the start-up and total costs are 0.00 and 145.00, respectively.

In this section, we will explore how to estimate the sequential scan, index scan, and sort operation in detail.

In the following explanations, we will use a specific table and an index that are shown below:

```
testdb=# CREATE TABLE tbl (id int PRIMARY KEY, data int);
testdb=# CREATE INDEX tbl_data_idx ON tbl (data);
testdb=# INSERT INTO tbl SELECT generate_series(1,10000),generate_series(1,10000);
testdb=# ANALYZE;
testdb=# \d tbl
Table "public.tbl"
```

Column	Type	Modifiers
id	integer	not null
data	integer	

Indexes:

- "tbl\_pkey" PRIMARY KEY, btree (id)
- "tbl\_data\_idx" btree (data)

### 3.2.1. Sequential Scan

The cost of the sequential scan is estimated by the `cost_seqscan()` function. In this subsection, we will explore how to estimate the sequential scan cost of the following query:

```
testdb=# SELECT * FROM tbl WHERE id < 8000;
```

In the sequential scan, the start-up cost is equal to 0, and the run cost is defined by the following equation:

$$\begin{aligned}\text{'run cost'} &= \text{'cpu run cost'} + \text{'disk run cost'} \\ &= (\text{cpu\_tuple\_cost} + \text{cpu\_operator\_cost}) \times N_{tuple} + \text{seq\_page\_cost} \times N_{page},\end{aligned}$$

where `seq_page_cost`, `cpu_tuple_cost` and `cpu_operator_cost` are set in the `postgresql.conf` file, and the default values are 1.0, 0.01, and 0.0025, respectively.  $N_{tuple}$  and  $N_{page}$  are the numbers of all tuples and all pages of this table, respectively. These numbers can be shown using the following query:

```
testdb=# SELECT relpages, reltuples FROM pg_class WHERE relname = 'tbl';
relpages | reltuples
-----+-----
45      |     10000
(1 row)
```

$$N_{tuple} = 10000, \quad (1)$$

$$N_{page} = 45. \quad (2)$$

Therefore,

$$\text{'run cost'} = (0.01 + 0.0025) \times 10000 + 1.0 \times 45 = 170.0.$$

Finally,

$$\text{'total cost'} = 0.0 + 170.0 = 170.$$

For confirmation, the result of the `EXPLAIN` command of the above query is shown below:

```
1. testdb=# EXPLAIN SELECT * FROM tbl WHERE id < 8000;
2.                                     QUERY PLAN
3.
4. Seq Scan on tbl  (cost=0.00..170.00 rows=8000 width=8)
5.   Filter: (id < 8000)
6.   (2 rows)
```

In Line 4, we can see that the start-up and total costs are 0.00 and 170.00, respectively. It is also estimated that 8000 rows (tuples) will be selected by scanning all rows.

In line 5, a filter 'Filter:(`id < 8000`)' of the sequential scan is shown. More precisely, it is called a *table level filter predicate*. Note that this type of filter is used when reading all the tuples in the

table, and it does not narrow the scanned range of table pages.



As understood from the run-cost estimation, PostgreSQL assumes that all pages will be read from storage. In other words, PostgreSQL does not consider whether the scanned page is in the shared buffers or not.

### 3.2.2. Index Scan

Although PostgreSQL supports some index methods, such as BTree, GiST, GIN and BRIN, the cost of the index scan is estimated using the common cost function `cost_index()`.

In this subsection, we explore how to estimate the index scan cost of the following query:

```
testdb=# SELECT id, data FROM tbl WHERE data < 240;
```

Before estimating the cost, the numbers of the index pages and index tuples,  $N_{index,page}$  and  $N_{index,tuple}$ , are shown below:

```
testdb=# SELECT relpages, reltuples FROM pg_class WHERE relname = 'tbl_data_idx';
          relpages |      reltuples
-----+-----
        30 |      10000
(1 row)
```

$$N_{index,tuple} = 10000, \quad (3)$$

$$N_{index,page} = 30. \quad (4)$$

#### 3.2.2.1. Start-Up Cost

The start-up cost of the index scan is the cost of reading the index pages to access the first tuple in the target table. It is defined by the following equation:

$$'start-up cost' = \{ceil(log_2(N_{index,tuple})) + (H_{index} + 1) \times 50\} \times \text{cpu\_operator\_cost},$$

where  $H_{index}$  is the height of the index tree.

In this case, according to (3),  $N_{index,tuple}$  is 10000,  $H_{index}$  is 1; `cpu_operator_cost` is 0.0025 (by default). Therefore,

$$'start-up cost' = \{ceil(log_2(10000)) + (1 + 1) \times 50\} \times 0.0025 = 0.285. \quad (5)$$

#### 3.2.2.2. Run Cost

The run cost of the index scan is the sum of the CPU costs and the I/O (input/output) costs of both the table and the index:

$$'run cost' = ('index cpu cost' + 'table cpu cost') + ('index IO cost' + 'table IO cost').$$



If the Index-Only Scans, which is described in Section 7.2, can be applied, 'table cpu cost' and 'table IO cost' are not estimated.

The first three costs (i.e., index CPU cost, table CPU cost, and index I/O cost) are shown below:

$$\begin{aligned}'\text{index cpu cost}' &= \text{Selectivity} \times N_{\text{index,tuple}} \times (\text{cpu\_index\_tuple\_cost} + \text{qual\_op\_cost}), \\ '\text{table cpu cost}' &= \text{Selectivity} \times N_{\text{tuple}} \times \text{cpu\_tuple\_cost}, \\ '\text{index IO cost}' &= \text{ceil}(\text{Selectivity} \times N_{\text{index,page}}) \times \text{random\_page\_cost},\end{aligned}$$

where

- `cpu_index_tuple_cost` and `random_page_cost` are set in the `postgresql.conf` file. The defaults are 0.005 and 4.0, respectively.
- `qual_op_cost` is, roughly speaking, the cost of evaluating the index predicate. The default is 0.0025.
- `Selectivity` is the proportion of the search range of the index that satisfies the WHERE clause, it is a floating-point number from 0 to 1. (`Selectivity`  $\times$  `Ntuple`) means *the number of the table tuples to be read*, (`Selectivity`  $\times$  `Nindex,page`) means *the number of the index pages to be read*.

Selectivity is described in detail in **i** below.

## **i** Selectivity

The selectivity of query predicates is estimated using either the `histogram_bounds` or the MCV (Most Common Value), both of which are stored in the statistics information in the `pg_stats`. Here, the calculation of the selectivity is briefly described using specific examples. More details are provided in the [official document](#).

The MCV of each column of a table is stored in the `pg_stats` view as a pair of columns named `most_common_vals` and `most_common_freqs`:

- `most_common_vals` is a list of the MCVs in the column.
- `most_common_freqs` is a list of the frequencies of the MCVs.

Here is a simple example: The table "countries" has two columns:

- a column 'country': This column stores the country name.
- a column 'continent': This column stores the continent name to which the country belongs.

```
testdb=# \d countries
  Table "public.countries"
 Column | Type | Modifiers
-----+-----+
 country | text |
 continent | text |
Indexes:
  "continent_idx" btree (continent)

testdb=# SELECT continent, count(*) AS "number of countries",
testdb#      (count(*)/(SELECT count(*) FROM countries)::real) AS "number of countries / all countries"
testdb#      FROM countries GROUP BY continent ORDER BY "number of countries" DESC;
   continent | number of countries | number of countries / all countries
-----+-----+-----+
 Africa      |          53 |        0.274611398963731
 Europe      |          47 |        0.243523316062176
 Asia         |          44 |        0.227979274611399
 North America |          23 |        0.119170984455959
 Oceania      |          14 |        0.0725388601036269
 South America |          12 |        0.0621761658031088
(6 rows)
```

Consider the following query, which has a WHERE clause, 'continent = 'Asia'':

```
testdb=# SELECT * FROM countries WHERE continent = 'Asia';
```

In this case, the planner estimates the index scan cost using the MCV of the 'continent' column. The `most\_common\_vals` and `most\_common\_freqs` of this column are shown below:

```
testdb=# \x
Expanded display is on.
testdb=# SELECT most_common_vals, most_common_freqs FROM pg_stats
testdb-# WHERE tablename = 'countries' AND attname='continent';
-[ RECORD 1 ]-----+
most_common_vals | {Africa,Europe,Asia, "North America",Oceania, "South America"}
most_common_freqs | {0.274611, 0.243523, 0.227979, 0.119171, 0.0725389, 0.0621762}
```

The value of most\_common\_freqs corresponding to 'Asia' of the most\_common\_vals is 0.227979. Therefore, 0.227979 is used as the selectivity in this estimation.

If the MCV cannot be used, e.g., the target column type is integer or double precision, then the value of the *histogram\_bounds* of the target column is used to estimate the cost.

- **histogram\_bounds** is a list of values that divide the column's values into groups of approximately equal population.

A specific example is shown. This is the value of the histogram\_bounds of the column 'data' in the table 'tbl':

```
testdb=# SELECT histogram_bounds FROM pg_stats WHERE tablename = 'tbl' AND attname = 'data';
histogram_bounds
-----
{1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100,
2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, 3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000, 4100,
4200, 4300, 4400, 4500, 4600, 4700, 4800, 4900, 5000, 5100, 5200, 5300, 5400, 5500, 5600, 5700, 5800, 5900, 6000, 6100,
6200, 6300, 6400, 6500, 6600, 6700, 6800, 6900, 7000, 7100, 7200, 7300, 7400, 7500, 7600, 7700, 7800, 7900, 8000, 8100,
8200, 8300, 8400, 8500, 8600, 8700, 8800, 8900, 9000, 9100, 9200, 9300, 9400, 9500, 9600, 9700, 9800, 9900, 10000}
(1 row)
```

By default, the histogram\_bounds is divided into 100 buckets. Figure 3.7 illustrates the buckets and the corresponding histogram\_bounds in this example. Buckets are numbered starting from 0, and every bucket stores (approximately) the same number of tuples. The values of histogram\_bounds are the bounds of the corresponding buckets. For example, the 0th value of the histogram bounds is 1, which means that it is the minimum value of the tuples stored in bucket 0. The 1st value is 100 and this is the minimum value of the tuples stored in bucket 1, and so on.

**Fig. 3.7. Buckets and histogram\_bounds.**

	bucket_0	bucket_1	bucket_2	bucket_3	.....	bucket_97	bucket_98	bucket_99	
histogram_bounds	hb(0) 1	hb(1) 100	hb(2) 200	hb(3) 300	hb(4)	hb(97) 9700	hb(98) 9800	hb(99) 9900	hb(100) 10000

Next, the calculation of the selectivity will be shown using the example in this subsection. The query has a WHERE clause 'data < 240' and the value '240' is in the second bucket. In this case, the selectivity can be derived by applying linear interpolation. Thus, the selectivity of the column 'data' in this query is calculated using the following equation:

$$\text{Selectivity} = \frac{2 + (240 - \text{hb}[2]) / (\text{hb}[3] - \text{hb}[2])}{100} = \frac{2 + (240 - 200) / (300 - 200)}{100} = \frac{2 + 40/100}{100} = 0.024. \quad (6)$$

Thus, according to (1),(3),(4) and (6),

$$\text{'index cpu cost'} = 0.024 \times 10000 \times (0.005 + 0.0025) = 1.8, \quad (7)$$

$$\text{'table cpu cost'} = 0.024 \times 10000 \times 0.01 = 2.4, \quad (8)$$

$$\text{'index IO cost'} = \text{ceil}(0.024 \times 30) \times 4.0 = 4.0. \quad (9)$$

'table IO cost' is defined by the following equation:

$$\text{'table IO cost'} = \text{max\_IO\_cost} + \text{indexCorrelation}^2 \times (\text{min\_IO\_cost} - \text{max\_IO\_cost}).$$

`max_IO_cost` is the worst case of the IO cost, that is, the cost of randomly scanning all table pages; this cost is defined by the following equation:

$$\text{max\_IO\_cost} = N_{\text{page}} \times \text{random\_page\_cost}.$$

In this case, according to (2),  $N_{\text{page}} = 45$ , and thus

$$\text{max\_IO\_cost} = 45 \times 4.0 = 180.0. \quad (10)$$

`min_IO_cost` is the best case of the IO cost, that is, the cost of sequentially scanning the selected table pages; this cost is defined by the following equation:

$$\text{min\_IO\_cost} = 1 \times \text{random\_page\_cost} + (\text{ceil}(\text{Selectivity} \times N_{\text{page}}) - 1) \times \text{seq\_page\_cost}.$$

In this case,

$$\text{min\_IO\_cost} = 1 \times 4.0 + (\text{ceil}(0.024 \times 45)) - 1) \times 1.0 = 5.0. \quad (11)$$

`indexCorrelation` is described in detail in [❶](#) below, and in this example,

$$\text{indexCorrelation} = 1.0. \quad (12)$$

Thus, according to (10),(11) and (12),

$$\text{'table IO cost'} = 180.0 + 1.0^2 \times (5.0 - 180.0) = 5.0. \quad (13)$$

Finally, according to (7),(8),(9) and (13),

$$\text{'run cost'} = (1.8 + 2.4) + (4.0 + 5.0) = 13.2. \quad (14)$$

## ❶ Index Correlation

Index correlation is a statistical correlation between the physical row ordering and the logical ordering of the column values (cited from the official document). This ranges from  $-1$  to  $+1$ . To understand the relation between the index scan and the index correlation, a specific example is shown below.

The table `tbl_corr` has five columns: two columns are text type and three columns are integer type. The three integer columns store numbers from 1 to 12. Physically, `tbl_corr` is composed of three pages, and each page has four tuples. Each integer type column has an index with a name such as `index_col_asc` and so on.

```
testdb=# \d tbl_corr
  Table "public.tbl_corr"
 Column | Type   | Modifiers
-----+-----+
 col    | text   |
 col_asc | integer |
 col_desc | integer |
 col_rand | integer |
 data   | text   |

Indexes:
 "tbl_corr_asc_idx" btree (col_asc)
 "tbl_corr_desc_idx" btree (col_desc)
```

```
"tbl_corr_rand_idx" btree (col_rand)
```

```
testdb=# SELECT col,col_asc,col_desc,col_rand
testdb-#                                     FROM tbl_corr;
   col | col_asc | col_desc | col_rand
-----+-----+-----+-----+
 Tuple_1 |      1 |      12 |      3
 Tuple_2 |      2 |      11 |      8
 Tuple_3 |      3 |      10 |      5
 Tuple_4 |      4 |      9 |      9
 Tuple_5 |      5 |      8 |      7
 Tuple_6 |      6 |      7 |      2
 Tuple_7 |      7 |      6 |     10
 Tuple_8 |      8 |      5 |     11
 Tuple_9 |      9 |      4 |      4
 Tuple_10 |     10 |      3 |      1
 Tuple_11 |     11 |      2 |     12
 Tuple_12 |     12 |      1 |      6
(12 rows)
```

The index correlations of these columns are shown below:

```
testdb=# SELECT tablename,attname, correlation FROM pg_stats WHERE tablename = 'tbl_corr';
 tablename | attname | correlation
-----+-----+-----+
tbl_corr | col_asc |      1
tbl_corr | col_desc |     -1
tbl_corr | col_rand | 0.125874
(3 rows)
```

When the following query is executed, PostgreSQL reads only the first page because all of the target tuples are stored in the first page. See Fig. 3.8(a).

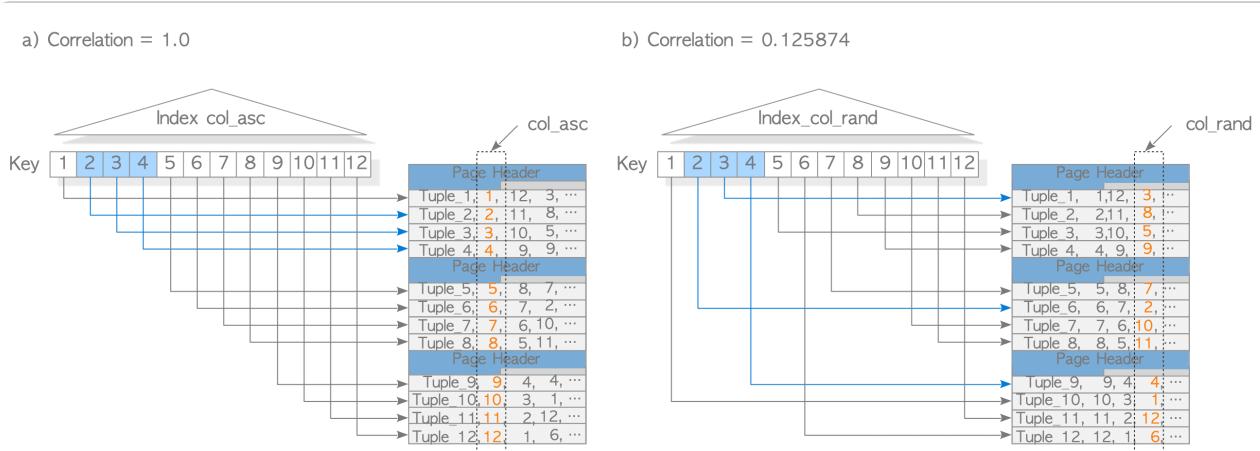
```
testdb=# SELECT * FROM tbl_corr WHERE col_asc BETWEEN 2 AND 4;
```

On the other hand, when the following query is executed, PostgreSQL has to read all pages. See Fig. 3.8(b).

```
testdb=# SELECT * FROM tbl_corr WHERE col_rand BETWEEN 2 AND 4;
```

In this way, the index correlation is a statistical correlation that reflects the impact of random access caused by the discrepancy between the index ordering and the physical tuple ordering in the table when estimating the index scan cost.

**Fig. 3.8. Index correlation.**



### 3.2.2.3. Total Cost

According to (3) and (14),

$$\text{'total cost'} = 0.285 + 13.2 = 13.485. \quad (15)$$

For confirmation, the result of the EXPLAIN command of the above SELECT query is shown below:

```
1. testdb=# EXPLAIN SELECT id, data FROM tbl WHERE data < 240;
2.                                     QUERY PLAN
3.
4. Index Scan using tbl_data_idx on tbl  (cost=0.29..13.49 rows=240 width=8)
5.   Index Cond: (data < 240)
6.   (2 rows)
```

In Line 4, we can find that the start-up and total costs are 0.29 and 13.49, respectively, and it is estimated that 240 rows (tuples) will be scanned.

In Line 5, an index condition 'Index Cond:(data < 240)' of the index scan is shown. More precisely, this condition is called an *access predicate*, and it expresses the start and stop conditions of the index scan.



According to this post, EXPLAIN command in PostgreSQL does not distinguish between the access predicate and index filter predicate. Therefore, if you analyze the output of EXPLAIN, pay attention not only to the index conditions but also to the estimated value of rows.

## ❶ Indexes Internals

This document does not explain indexes in details. To understand them, I recommend to read the valuable posts shown below:

- Indexes in PostgreSQL — 1
- Indexes in PostgreSQL — 2
- Indexes in PostgreSQL — 3 (Hash)
- Indexes in PostgreSQL — 4 (Btree)
- Indexes in PostgreSQL — 5 (GiST)
- Indexes in PostgreSQL — 6 (SP-GiST)
- Indexes in PostgreSQL — 7 (GIN)
- Indexes in PostgreSQL — 9 (BRIN)

## ❶ seq\_page\_cost and random\_page\_cost

The default values of `seq_page_cost` and `random_page_cost` are 1.0 and 4.0, respectively. This means that PostgreSQL assumes that the random scan is four times slower than the sequential scan. In other words, the default value of PostgreSQL is based on using HDDs.

On the other hand, in recent days, the default value of random\_page\_cost is too large because SSDs are mostly used. If the default value of random\_page\_cost is used despite using an SSD, the planner may select ineffective plans. Therefore, when using an SSD, it is better to change the value of random\_page\_cost to 1.0.

This blog reported the problem when using the default value of random\_page\_cost.

### 3.2.3. Sort

The sort path is used for sorting operations, such as ORDER BY, the preprocessing of merge join operations, and other functions. The cost of sorting is estimated using the cost\_sort() function.

In the sorting operation, if all tuples to be sorted can be stored in work\_mem, the quicksort algorithm is used. Otherwise, a temporary file is created and the file merge sort algorithm is used.

The start-up cost of the sort path is the cost of sorting the target tuples. Therefore, the cost is  $O(N_{sort} \times \log_2(N_{sort}))$ , where  $N_{sort}$  is the number of the tuples to be sorted. The run cost of the sort path is the cost of reading the sorted tuples. Therefore the cost is  $O(N_{sort})$ .

In this subsection, we explore how to estimate the sorting cost of the following query. Assume that this query will be sorted in work\_mem, without using temporary files.

```
testdb=# SELECT id, data FROM tbl WHERE data < 240 ORDER BY id;
```

In this case, the start-up cost is defined in the following equation:

$$\text{'start-up cost'} = C + \text{comparison\_cost} \times N_{sort} \times \log_2(N_{sort}),$$

where

- $C$  is the total cost of the last scan, that is, the total cost of the index scan; according to (15), it is 13.485.
- $N_{sort}$  is the number of tuples to be sorted. In this case, it is 240.
- `comparison_cost` is defined in  $2 \times \text{cpu\_operator\_cost}$ .

Therefore, the start-up cost is calculated as follows:

$$\text{'start-up cost'} = 13.485 + (2 \times 0.0025) \times 240.0 \times \log_2(240.0) = 22.973.$$

The run cost is the cost of reading sorted tuples in the memory. Therefore,

$$\text{'run cost'} = \text{cpu\_operator\_cost} \times N_{sort} = 0.0025 \times 240 = 0.6.$$

Finally,

$$\text{'total cost'} = 22.973 + 0.6 = 23.573.$$

For confirmation, the result of the EXPLAIN command of the above SELECT query is shown below:

```
1. testdb=# EXPLAIN SELECT id, data FROM tbl WHERE data < 240 ORDER BY id;
2.                                     QUERY PLAN
3. -----
4.   Sort  (cost=22.97..23.57 rows=240 width=8)
5.     Sort Key: id
6.     -> Index Scan using tbl_data_idx on tbl  (cost=0.29..13.49 rows=240 width=
8)
```

7. Index Cond: (data < 240)  
8. (4 rows)

In line 4, we can find that the start-up cost and total cost are 22.97 and 23.57, respectively.

### 3.3. Creating the Plan Tree of a Single-Table Query

As the processing of the planner is very complicated, this section describes the simplest process, namely, how a plan tree of a single-table query is created. More complex processing, namely, how a plan tree of a multi-table query is created, is described in Section 3.6.

The planner in PostgreSQL performs three steps, as shown below:

1. Carry out preprocessing.
2. Get the cheapest access path by estimating the costs of all possible access paths.
3. Create the plan tree from the cheapest path.

An access path is a unit of processing for estimating the cost. For example, the sequential scan, index scan, sort, and various join operations have their corresponding paths. Access paths are used only inside the planner to create the plan tree. The most fundamental data structure of access paths is the Path structure defined in `pathnodes.h`, and it corresponds to the sequential scan. All other access paths are based on it. Details will be described in the following explanations.

```
typedef struct PathKey
{
    pg_node_attr(no_read, no_query_jumble)
    NodeTag           type;

    /* the value that is ordered */
    EquivalenceClass *pk_eclass pg_node_attr(copy_as_scalar, equal_as_scalar);
    Oid              pk_opfamily;    /* btree opfamily defining the ordering */
    int               pk_strategy;   /* sort direction (ASC or DESC) */
    bool              pk_nulls_first; /* do NULLs come before normal values? */
} PathKey;

typedef struct Path
{
    pg_node_attr(no_copy_equal, no_read, no_query_jumble)
    NodeTag           type;

    /* tag identifying scan/join method */
    NodeTag           pathtype;

    /*
     * the relation this path can build
     *
     * We do NOT print the parent, else we'd be in infinite recursion.  We can
     * print the parent's relids for identification purposes, though.
     */
    RelOptInfo *parent pg_node_attr(write_only_relids);

    /*
     * list of Vars/Exprs, cost, width
     *
     * We print the pathtarget only if it's not the default one for the rel.
    
```

```

/*
PathTarget *pathtarget pg_node_attr(write_only_nondefault_pathtarget);

/*
 * parameterization info, or NULL if none
 *
 * We do not print the whole of param_info, since it's printed via
 * RelOptInfo; it's sufficient and less cluttering to print just the
 * required outer relids.
*/
ParamPathInfo *param_info pg_node_attr(write_only_req_outer);

/* engage parallel-aware logic? */
bool parallel_aware;
/* OK to use as part of parallel plan? */
bool parallel_safe;
/* desired # of workers; 0 = not parallel */
int parallel_workers;

/* estimated size/costs for path (see costsize.c for more info) */
Cardinality rows; /* estimated number of result tuples
*/
Cost startup_cost; /* cost expended before fetching any tuples
*/
Cost total_cost; /* total cost (assuming all tuples fetched) */

/* sort ordering of path's output; a List of PathKey nodes; see above */
List *pathkeys;
} Path;
}

```

To process the above steps, the planner internally creates a `PlannerInfo` structure, and holds the query tree, the information about the relations contained in the query, the access paths, and so on.

```

/*-----
 * PlannerInfo
 * Per-query information for planning/optimization
 *
 * This struct is conventionally called "root" in all the planner routines.
 * It holds links to all of the planner's working state, in addition to the
 * original Query. Note that at present the planner extensively modifies
 * the passed-in Query data structure; someday that should stop.
 *
 * For reasons explained in optimizer/optimizer.h, we define the typedef
 * either here or in that header, whichever is read first.
 *
 * Not all fields are printed. (In some cases, there is no print support for
 * the field type; in others, doing so would lead to infinite recursion or
 * bloat dump output more than seems useful.)
*-----
*/
#ifndef HAVE_PLANNERINFO_TYPEDEF
typedef struct PlannerInfo PlannerInfo;
#define HAVE_PLANNERINFO_TYPEDEF 1
#endif

struct PlannerInfo
{
    pg_node_attr(no_copy_equal, no_read, no_query_jumble)

    NodeTag type;
}

```

```

/* the Query being planned */
Query      *parse;

/* global info for current planner run */
PlannerGlobal *glob;

/* 1 at the outermost Query */
Index          query_level;

/* NULL at outermost Query */
PlannerInfo *parent_root pg_node_attr(read_write_ignore);

/*
 * plan_params contains the expressions that this query level needs to
 * make available to a lower query level that is currently being planned.
 * outer_params contains the paramIds of PARAM_EXEC Params that outer
 * query levels will make available to this query level.
 */
/* list of PlannerParamItems, see below */
List      *plan_params;
Bitmapset *outer_params;

/*
 * simple_rel_array holds pointers to "base rels" and "other rels" (see
 * comments for RelOptInfo for more info). It is indexed by rangetable
 * index (so entry 0 is always wasted). Entries can be NULL when an RTE
 * does not correspond to a base relation, such as a join RTE or an
 * unreferenced view RTE; or if the RelOptInfo hasn't been made yet.
 */
struct RelOptInfo **simple_rel_array pg_node_attr(array_size(simple_rel_ar
ray_size));
/* allocated size of array */
int           simple_rel_array_size;

/*
 * simple_rte_array is the same length as simple_rel_array and holds
 * pointers to the associated rangetable entries. Using this is a shade
 * faster than using rt_fetch(), mostly due to fewer indirections. (Not
 * printed because it'd be redundant with parse->rtable.)
 */
RangeTblEntry **simple_rte_array pg_node_attr(read_write_ignore);

/*
 * append_rel_array is the same length as the above arrays, and holds
 * pointers to the corresponding AppendRelInfo entry indexed by
 * child_relid, or NULL if the rel is not an appendrel child. The array
 * itself is not allocated if append_rel_list is empty. (Not printed
 * because it'd be redundant with append_rel_list.)
 */
struct AppendRelInfo **append_rel_array pg_node_attr(read_write_ignore);

/*
 * all_baserels is a Relids set of all base relids (but not joins or
 * "other" rels) in the query. This is computed in deconstruct_jointree.
 */
Relids       all_baserels;

/*
 * outer_join_rels is a Relids set of all outer-join relids in the query.
 * This is computed in deconstruct_jointree.
*/

```

```

*/
Relids      outer_join_rels;

/*
 * all_query_rels is a Relids set of all base relids and outer join relids
 * (but not "other" relids) in the query. This is the Relids identifier
 * of the final join we need to form. This is computed in
 * deconstruct_jointree.
*/
Relids      all_query_rels;

/*
 * join_rel_list is a list of all join-relation RelOptInfos we have
 * considered in this planning run. For small problems we just scan the
 * list to do lookups, but when there are many join relations we build a
 * hash table for faster lookups. The hash table is present and valid
 * when join_rel_hash is not NULL. Note that we still maintain the list
 * even when using the hash table for lookups; this simplifies life for
 * GEQO.
*/
List      *join_rel_list;
struct HTAB *join_rel_hash pg_node_attr(read_write_ignore);

/*
 * When doing a dynamic-programming-style join search, join_rel_level[k]
 * is a list of all join-relation RelOptInfos of level k, and
 * join_cur_level is the current level. New join-relation RelOptInfos are
 * automatically added to the join_rel_level[join_cur_level] list.
 * join_rel_level is NULL if not in use.
 *
 * Note: we've already printed all baserel and joinrel RelOptInfos above,
 * so we don't dump join_rel_level or other lists of RelOptInfos.
 */
/* lists of join-relation RelOptInfos */
List      **join_rel_level pg_node_attr(read_write_ignore);
/* index of list being extended */
int      join_cur_level;

/* init SubPlans for query */
List      *init_plans;

/*
 * per-CTE-item list of subplan IDs (or -1 if no subplan was made for that
 * CTE)
 */
List      *cte_plan_ids;

/* List of Lists of Params for MULTIEXPR subquery outputs */
List      *multiexpr_params;

/* list of JoinDomains used in the query (higher ones first) */
List      *join_domains;

/* list of active EquivalenceClasses */
List      *eq_classes;

/* set true once ECs are canonical */
bool      ec_merging_done;

/* list of "canonical" PathKeys */
List      *canon_pathkeys;

```

```

/*
 * list of OuterJoinClauseInfos for mergejoinable outer join clauses
 * w/nonnullable var on left
 */
List      *left_join_clauses;

/*
 * list of OuterJoinClauseInfos for mergejoinable outer join clauses
 * w/nonnullable var on right
 */
List      *right_join_clauses;

/*
 * list of OuterJoinClauseInfos for mergejoinable full join clauses
 */
List      *full_join_clauses;

/* list of SpecialJoinInfos */
List      *join_info_list;

/* counter for assigning RestrictInfo serial numbers */
int          last_rinfo_serial;

/*
 * all_result_relids is empty for SELECT, otherwise it contains at least
 * parse->resultRelation. For UPDATE/DELETE/MERGE across an inheritance
 * or partitioning tree, the result rel's child relids are added. When
 * using multi-level partitioning, intermediate partitioned rels are
 * included. leaf_result_relids is similar except that only actual result
 * tables, not partitioned tables, are included in it.
 */
Relids      all_result_relids;
/* set of all leaf relids */
Relids      leaf_result_relids;

/*
 * list of AppendRelInfos
 *
 * Note: for AppendRelInfos describing partitions of a partitioned table,
 * we guarantee that partitions that come earlier in the partitioned
 * table's PartitionDesc will appear earlier in append_rel_list.
 */
List      *append_rel_list;

/* list of RowIdentityVarInfos */
List      *row_identity_vars;

/* list of PlanRowMarks */
List      *rowMarks;

/* list of PlaceHolderInfos */
List      *placeholder_list;

/* array of PlaceHolderInfos indexed by phid */
struct PlaceHolderInfo **placeholder_array pg_node_attr(read_write_ignore,
array_size(placeholder_array_size));
/* allocated size of array */
int          placeholder_array_size pg_node_attr(read_write_ignore
e);

```

```

/* list of ForeignKeyOptInfos */
List *fkey_list;

/* desired pathkeys for query_planner() */
List *query_pathkeys;

/* groupClause pathkeys, if any */
List *group_pathkeys;

/*
 * The number of elements in the group_pathkeys list which belong to the
 * GROUP BY clause. Additional ones belong to ORDER BY / DISTINCT
 * aggregates.
 */
int num_groupby_pathkeys;

/* pathkeys of bottom window, if any */
List *window_pathkeys;
/* distinctClause pathkeys, if any */
List *distinct_pathkeys;
/* sortClause pathkeys, if any */
List *sort_pathkeys;

/* Canonicalised partition schemes used in the query. */
List *part_schemes pg_node_attr(read_write_ignore);

/* RelOptInfos we are now trying to join */
List *initial_rels pg_node_attr(read_write_ignore);

/*
 * Upper-rel RelOptInfos. Use fetch_upper_rel() to get any particular
 * upper rel.
 */
List *upper_rels[UPPERREL_FINAL + 1] pg_node_attr(read_write_ignore);

/* Result tlists chosen by grouping_planner for upper-stage processing */
struct PathTarget *upper_targets[UPPERREL_FINAL + 1] pg_node_attr(read_write_ignore);

/*
 * The fully-processed groupClause is kept here. It differs from
 * parse->groupClause in that we remove any items that we can prove
 * redundant, so that only the columns named here actually need to be
 * compared to determine grouping. Note that it's possible for *all* the
 * items to be proven redundant, implying that there is only one group
 * containing all the query's rows. Hence, if you want to check whether
 * GROUP BY was specified, test for nonempty parse->groupClause, not for
 * nonempty processed_groupClause.
 *
 * Currently, when grouping sets are specified we do not attempt to
 * optimize the groupClause, so that processed_groupClause will be
 * identical to parse->groupClause.
 */
List *processed_groupClause;

/*
 * The fully-processed distinctClause is kept here. It differs from
 * parse->distinctClause in that we remove any items that we can prove
 * redundant, so that only the columns named here actually need to be
 * compared to determine uniqueness. Note that it's possible for *all*

```

```

* the items to be proven redundant, implying that there should be only
* one output row. Hence, if you want to check whether DISTINCT was
* specified, test for nonempty parse->distinctClause, not for nonempty
* processed_distinctClause.
*/
List      *processed_distinctClause;

/*
* The fully-processed targetlist is kept here. It differs from
* parse->targetList in that (for INSERT) it's been reordered to match the
* target table, and defaults have been filled in. Also, additional
* resjunk targets may be present. preprocess_targetlist() does most of
* that work, but note that more resjunk targets can get added during
* appendrel expansion. (Hence, upper_targets mustn't get set up till
* after that.)
*/
List      *processed_tlist;

/*
* For UPDATE, this list contains the target table's attribute numbers to
* which the first N entries of processed_tlist are to be assigned. (Any
* additional entries in processed_tlist must be resjunk.) DO NOT use the
* resnos in processed_tlist to identify the UPDATE target columns.
*/
List      *update_colnos;

/*
* Fields filled during create_plan() for use in setrefs.c
*/
/* for GroupingFunc fixup (can't print: array length not known here) */
AttrNumber *grouping_map pg_node_attr(read_write_ignore);
/* List of MinMaxAggInfos */
List      *minmax_aggs;

/* context holding PlannerInfo */
MemoryContext planner_ctxt pg_node_attr(read_write_ignore);

/* # of pages in all non-dummy tables of query */
Cardinality total_table_pages;

/* tuple_fraction passed to query_planner */
Selectivity tuple_fraction;
/* limit_tuples passed to query_planner */
Cardinality limit_tuples;

/*
* Minimum security_level for quals. Note: qual_security_level is zero if
* there are no securityQuals.
*/
Index      qual_security_level;

/* true if any RTEs are RTE_JOIN kind */
bool      hasJoinRTEs;
/* true if any RTEs are marked LATERAL */
bool      hasLateralRTEs;
/* true if havingQual was non-null */
bool      hasHavingQual;
/* true if any RestrictInfo has pseudoconstant = true */
bool      hasPseudoConstantQuals;
/* true if we've made any of those */
bool      hasAlternativeSubPlans;

```

```

/* true once we're no longer allowed to add PlaceHolderInfos */
bool          placeholdersFrozen;
/* true if planning a recursive WITH item */
bool          hasRecursion;

/*
 * Information about aggregates. Filled by preprocess_aggregrefs().
 */
/* AggInfo structs */
List      *agginfos;
/* AggTransInfo structs */
List      *aggtransinfos;
/* number of aggs with DISTINCT/ORDER BY/WITHIN GROUP */
int       numOrderedAggs;
/* does any agg not support partial mode? */
bool      hasNonPartialAggs;
/* is any partial agg non-serializable? */
bool      hasNonSerialAggs;

/*
 * These fields are used only when hasRecursion is true:
 */
/* PARAM_EXEC ID for the work table */
int       wt_param_id;
/* a path for non-recursive term */
struct Path *non_recursive_path;

/*
 * These fields are workspace for createplan.c
 */
/* outer rels above current node */
Relids   curOuterRels;
/* not-yet-assigned NestLoopParams */
List      *curOuterParams;

/*
 * These fields are workspace for setrefs.c. Each is an array
 * corresponding to glob->subplans. (We could probably teach
 * gen_node_support.pl how to determine the array length, but it doesn't
 * seem worth the trouble, so just mark them read_write_ignore.)
 */
bool      *isAltSubplan pg_node_attr(read_write_ignore);
bool      *isUsedSubplan pg_node_attr(read_write_ignore);

/* optional private data for join_search_hook, e.g., GEQO */
void      *join_search_private pg_node_attr(read_write_ignore);

/* Does this query modify any partition key columns? */
bool      partColsUpdated;
};


```

In this section, how plan trees are created from query trees is described using specific examples.

### 3.3.1. Preprocessing

Before creating a plan tree, the planner carries out some preprocessing of the query tree stored in the PlannerInfo structure.

Although preprocessing involves many steps, we only discuss the main preprocessing for the single-table query in this subsection. The other preprocessing operations are described in Section

### 3.6.

The preprocessing steps include:

1. Simplifying target lists, limit clauses, and so on.

For example, the `eval_const_expressions()` function defined in `clauses.c` rewrites '`2 + 2`' to '`4`'.

2. Normalizing Boolean expressions.

For example, '`NOT (NOT a)`' is rewritten to '`a`'.

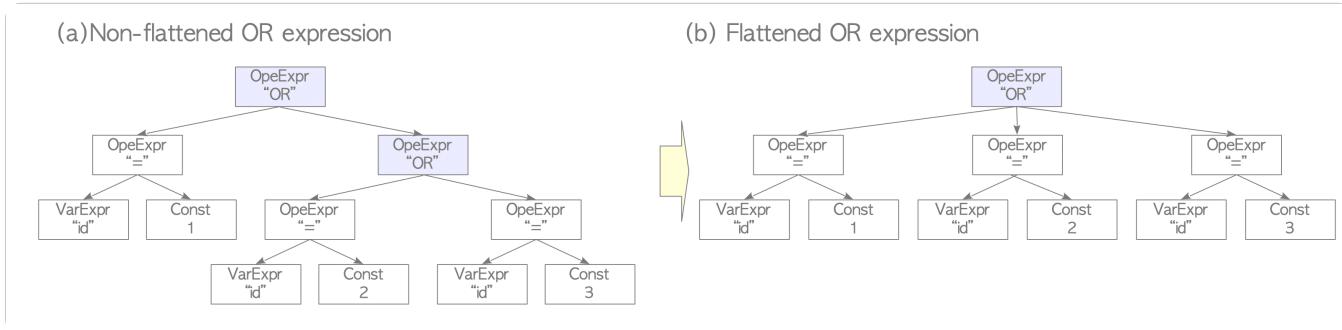
3. Flattening AND/OR expressions.

AND and OR in the SQL standard are binary operators, but in PostgreSQL internals, they are n-ary operators and the planner always assumes that all nested AND and OR expressions are to be flattened.

A specific example is shown. Consider a Boolean expression '`(id = 1) OR (id = 2) OR (id = 3)`'.

Figure 3.9(a) shows part of the query tree when using the binary operator. The planner simplified this tree by flattening using a ternary operator. See Fig. 3.9(b).

**Fig. 3.9. An example of flattening AND/OR expressions.**



### 3.3.2. Getting the Cheapest Access Path

To get the cheapest access path, the planner estimates the costs of all possible access paths and chooses the cheapest one. More specifically, the planner performs the following operations:

1. Create a `RelOptInfo` structure to store the access paths and the corresponding costs.

A `RelOptInfo` structure is created by the `make_one_rel()` function and is stored in the `simple_rel_array` of the `PlannerInfo` structure. See Fig. 3.10. In its initial state, the `RelOptInfo` holds the `baserestrictinfo` and the `indexlist` if related indexes exist. The `baserestrictinfo` stores the WHERE clauses of the query, and the `indexlist` stores the related indexes of the target table.

```
typedef enum RelOptKind
{
    RELOPT_BASEREL,
    RELOPT_JOINREL,
    RELOPT_OTHER_MEMBER_REL,
    RELOPT_OTHER_JOINREL,
    RELOPT_UPPER_REL,
    RELOPT_OTHER_UPPER_REL
} RelOptKind;

/*
 * Is the given relation a simple relation i.e a base or "other" member
 * relation?
 */
#define IS_SIMPLE_REL(rel) \
((rel)->reloptkind == RELOPT_BASEREL || \
```

```

(rel)->reloptkind == RELOPT_OTHER_MEMBER_REL)

/* Is the given relation a join relation? */
#define IS_JOIN_REL(rel) \
    ((rel)->reloptkind == RELOPT_JOINREL || \
     (rel)->reloptkind == RELOPT_OTHER_JOINREL)

/* Is the given relation an upper relation? */
#define IS_UPPER_REL(rel) \
    ((rel)->reloptkind == RELOPT_UPPER_REL || \
     (rel)->reloptkind == RELOPT_OTHER_UPPER_REL)

/* Is the given relation an "other" relation? */
#define IS_OTHER_REL(rel) \
    ((rel)->reloptkind == RELOPT_OTHER_MEMBER_REL || \
     (rel)->reloptkind == RELOPT_OTHER_JOINREL || \
     (rel)->reloptkind == RELOPT_OTHER_UPPER_REL)

typedef struct RelOptInfo
{
    pg_node_attr(no_copy_equal, no_read, no_query_jumble)

    NodeTag           type;

    RelOptKind        reloptkind;

    /*
     * all relations included in this RelOptInfo; set of base + OJ relids
     * (rangetable indexes)
     */
    Relids            relids;

    /*
     * size estimates generated by planner
     */
    /* estimated number of result tuples */
    Cardinality      rows;

    /*
     * per-relation planner control flags
     */
    /* keep cheap-startup-cost paths? */
    bool              consider_startup;
    /* ditto, for parameterized paths? */
    bool              consider_param_startup;
    /* consider parallel paths? */
    bool              consider_parallel;

    /*
     * default result targetlist for Paths scanning this relation; list of
     * Vars/Exprs, cost, width
     */
    struct PathTarget *reltarget;

    /*
     * materialization information
     */
    List              *pathlist;          /* Path structures */
    List              *ppilist;           /* ParamPathInfos used in pathlist */
    List              *partial_pathlist; /* partial Paths */
    struct Path *cheapest_startup_path;
}

```

```

    struct Path *cheapest_total_path;
    struct Path *cheapest_unique_path;
    List      *cheapest_parameterized_paths;

    /*
     * parameterization information needed for both base rels and join rels
     * (see also lateral_vars and lateral_referencers)
     */
    /* rels directly laterally referenced */
    Relids      direct_lateral_relids;
    /* minimum parameterization of rel */
    Relids      lateral_relids;

    /*
     * information about a base rel (not set for join rels!)
     */
    Index      relid;
    /* containing tablespace */
    Oid       reltablespace;
    /* RELATION, SUBQUERY, FUNCTION, etc */
    RTEKind    rtekind;
    /* smallest attrno of rel (often <0) */
    AttrNumber min_attr;
    /* largest attrno of rel */
    AttrNumber max_attr;
    /* array indexed [min_attr .. max_attr] */
    Relids    *attr_needed pg_node_attr(read_write_ignore);
    /* array indexed [min_attr .. max_attr] */
    int32     *attr_widths pg_node_attr(read_write_ignore);
    /* relids of outer joins that can null this baserel */
    Relids    nulling_relids;
    /* LATERAL Vars and PHVs referenced by rel */
    List      *lateral_vars;
    /* rels that reference this baserel laterally */
    Relids    lateral_referencers;
    /* list of IndexOptInfo */
    List      *indexlist;
    /* list of StatisticExtInfo */
    List      *statlist;
    /* size estimates derived from pg_class */
    BlockNumber pages;
    Cardinality tuples;
    double     allvisfrac;
    /* indexes in PlannerInfo's eq_classes list of ECs that mention this rel
     */
    Bitmapset *eclasse_indexes;
    PlannerInfo *subroot;           /* if subquery */
    List      *subplan_params; /* if subquery */
    /* wanted number of parallel workers */
    int       rel_parallel_workers;
    /* Bitmask of optional features supported by the table AM */
    uint32    amflags;

    /*
     * Information about foreign tables and foreign joins
     */
    /* identifies server for the table or join */
    Oid       serverid;
    /* identifies user to check access as; 0 means to check as current user
     */
    Oid       userid;

```

```

/* join is only valid for current user */
bool          useridiscurrent;
/* use "struct FdwRoutine" to avoid including fdwapi.h here */
struct FdwRoutine *fdwroutine pg_node_attr(read_write_ignore);
void          *fdw_private pg_node_attr(read_write_ignore);

/*
 * cache space for remembering if we have proven this relation unique
 */
/* known unique for these other relid set(s) */
List          *unique_for_rels;
/* known not unique for these set(s) */
List          *non_unique_for_rels;

/*
 * used by various scans and joins:
 */
/* RestrictInfo structures (if base rel) */
List          *baserestrictinfo;
/* cost of evaluating the above */
QualCost      baserestrictcost;
/* min security_level found in baserestrictinfo */
Index         baserestrict_min_security;
/* RestrictInfo structures for join clauses involving this rel */
List          *joininfo;
/* T means joininfo is incomplete */
bool          has_eclasse_joins;

/*
 * used by partitionwise joins:
 */
/* consider partitionwise join paths? (if partitioned rel) */
bool          consider_partitionwise_join;

/*
 * inheritance links, if this is an otherrel (otherwise NULL):
 */
/* Immediate parent relation (dumping it would be too verbose) */
struct RelOptInfo *parent pg_node_attr(read_write_ignore);
/* Topmost parent relation (dumping it would be too verbose) */
struct RelOptInfo *top_parent pg_node_attr(read_write_ignore);
/* Relids of topmost parent (redundant, but handy) */
Relids        top_parent_relids;

/*
 * used for partitioned relations:
 */
/* Partitioning scheme */
PartitionScheme part_scheme pg_node_attr(read_write_ignore);

/*
 * Number of partitions; -1 if not yet set; in case of a join relation 0
 * means it's considered unpartitioned
 */
int           nparts;
/* Partition bounds */
struct PartitionBoundInfoData *boundinfo pg_node_attr(read_write_ignore);
/* True if partition bounds were created by partition_bounds_merge() */
bool          partbounds_merged;
/* Partition constraint, if not the root */

```

```

List      *partition_qual;

/*
 * Array of RelOptInfos of partitions, stored in the same order as bounds
 * (don't print, too bulky and duplicative)
 */
struct RelOptInfo **part_rels pg_node_attr(read_write_ignore);

/*
 * Bitmap with members acting as indexes into the part_rels[] array to
 * indicate which partitions survived partition pruning.
 */
Bitmapset *live_parts;
/* Relids set of all partition relids */
Relids      all_partrels;

/*
 * These arrays are of length partkey->partnatts, which we don't have at
 * hand, so don't try to print
 */

/* Non-nullable partition key expressions */
List      **partexprs pg_node_attr(read_write_ignore);
/* Nullable partition key expressions */
List      **nullable_partexprs pg_node_attr(read_write_ignore);
} RelOptInfo;

```

2. Estimate the costs of all possible access paths, and add the access paths to the RelOptInfo structure.

Details of this processing are as follows:

1. A path is created, the cost of the sequential scan is estimated, and the estimated costs are written to the path. Then, the path is added to the pathlist of the RelOptInfo structure.
2. If indexes related to the target table exist, index access paths are created, all index scan costs are estimated, and the estimated costs are written to the path. Then, the index paths are added to the pathlist.
3. If the bitmap scan can be done, bitmap scan paths are created, all bitmap scan costs are estimated, and the estimated costs are written to the path. Then, the bitmap scan paths are added to the pathlist.
3. Get the cheapest access path in the pathlist of the RelOptInfo structure.
4. Estimate LIMIT, ORDER BY and ARREGISFDD costs if necessary.

To understand how the planner performs clearly, two specific examples are shown below.

### 3.3.2.1. Example 1

First, we explore a simple-single table query without indexes; this query contains both WHERE and ORDER BY clauses.

```

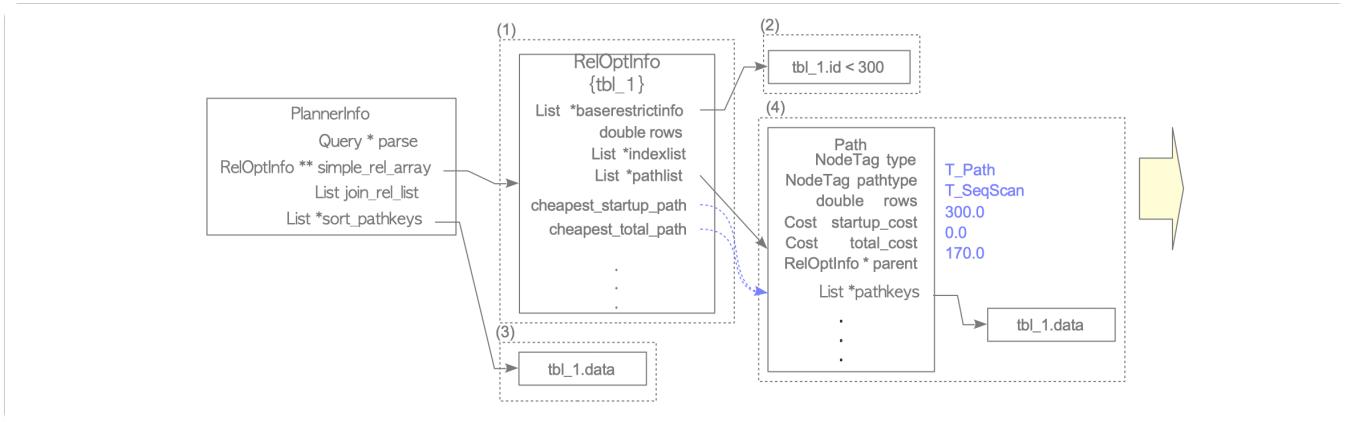
testdb=# \d tbl_1
Table "public.tbl_1"
Column | Type   | Modifiers
-----+-----+
id    | integer |
data  | integer |

testdb=# SELECT * FROM tbl_1 WHERE id < 300 ORDER BY data;

```

Figures 3.10 and 3.11 depict how the planner performs in this example.

**Fig. 3.10. How to get the cheapest path of Example 1.**



(1) Create a RelOptInfo structure and store it in the simple\_rel\_array of the PlannerInfo.

(2) Add a WHERE clause to the baserestrictinfo of the RelOptInfo.

A WHERE clause '`id < 300`' is added to the baserestrictinfo by the `distribute_restrictinfo_to_rels()` function defined in `initsplan.c`. In addition, the indexlist of the RelOptInfo is NULL because there are no related indexes of the target table.

(3) Add the pathkey for sorting to the sort\_pathkeys of the PlannerInfo by the `standard_qp_callback()` function defined in `planner.c`.

*Pathkey* is a data structure representing the sort ordering for the path. In this example, the column "data" is added to the sort\_pathkeys as a pathkey because this query contains an ORDER BY clause and its column is 'data'.

(4) Create a path structure and estimate the cost of the sequential scan using the `cost_seqscan` function and write the estimated costs into the path. Then, add the path to the RelOptInfo by the `add_path()` function defined in `pathnode.c`.

As mentioned before, the Path structure contains both of the start-up and the total costs which are estimated by the `cost_seqscan` function, and so on.

```

typedef struct PathKey
{
    pg_node_attr(no_read, no_query_jumble)
    NodeTag           type;

    /* the value that is ordered */
    EquivalenceClass *pk_eclass pg_node_attr(copy_as_scalar, equal_as_scalar);
    Oid               pk_opfamily; /* btree opfamily defining the ordering */
    int               pk_strategy; /* sort direction (ASC or DESC) */
    bool              pk_nulls_first; /* do NULLs come before normal values? */
} PathKey;

typedef struct Path
{
    pg_node_attr(no_copy_equal, no_read, no_query_jumble)
    NodeTag           type;

    /* tag identifying scan/join method */
    NodeTag           pathtype;

    /*
     * the relation this path can build
     */
}

```

```

* We do NOT print the parent, else we'd be in infinite recursion. We can
* print the parent's relids for identification purposes, though.
*/
RelOptInfo *parent pg_node_attr(write_only_relids);

/*
 * list of Vars/Exprs, cost, width
 *
 * We print the pathtarget only if it's not the default one for the rel.
 */
PathTarget *pathtarget pg_node_attr(write_only_nondefault_pathtarget);

/*
 * parameterization info, or NULL if none
 *
 * We do not print the whole of param_info, since it's printed via
 * RelOptInfo; it's sufficient and less cluttering to print just the
 * required outer relids.
 */
ParamPathInfo *param_info pg_node_attr(write_only_req_outer);

/* engage parallel-aware logic? */
bool parallel_aware;
/* OK to use as part of parallel plan? */
bool parallel_safe;
/* desired # of workers; 0 = not parallel */
int parallel_workers;

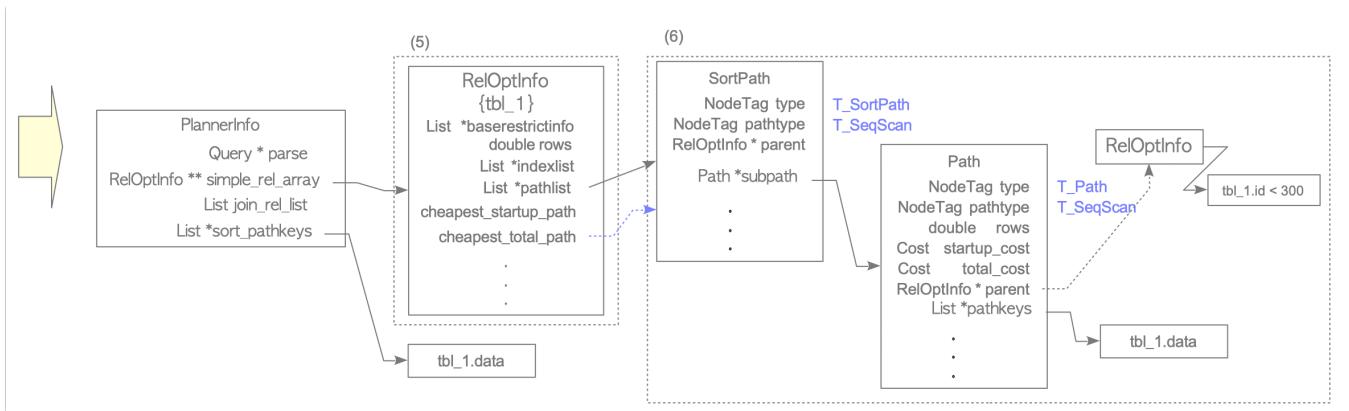
/* estimated size/costs for path (see costsize.c for more info) */
Cardinality rows; /* estimated number of result tuples
*/
Cost startup_cost; /* cost expended before fetching any tuples
*/
Cost total_cost; /* total cost (assuming all tuples fe-
tched) */

/* sort ordering of path's output; a List of PathKey nodes; see above */
List *pathkeys;
} Path;
}

```

In this example, the planner only estimates the sequential scan cost because there are no indexes of the target table. Therefore, the cheapest access path is automatically determined.

**Fig. 3.11. How to get the cheapest path of Example 1 (continued from Fig. 3.10).**



(5) Create a new RelOptInfo structure to process the ORDER BY procedure.

Note that the new RelOptInfo does not have the baserestrictinfo, that is, the information of the WHERE clause.

- (6) Create a sort path and add it to the new RelOptInfo; then, link the sequential scan path to the subpath of the sort path.

The SortPath structure is composed of two path structures: path and subpath; the path stores information about the sort operation itself, and the subpath stores the cheapest path.

Note that the item 'parent' of the sequential scan path holds the link to the old RelOptInfo which stores the WHERE clause in its baserestrictinfo. Therefore, in the next stage, that is, creating a plan tree, the planner can create a sequential scan node that contains the WHERE clause as the 'Filter', even though the new RelOptInfo does not have the baserestrictinfo.

```
typedef struct SortPath
{
    Path    path;
    Path    *subpath;           /* path representing input source */
} SortPath;
```

Based on the cheapest access path obtained here, a plan tree is generated. Details are described in Section 3.3.3.

### 3.3.2.2. Example 2

Next, we explore another single-table query with two indexes; this query contains a WHERE clause.

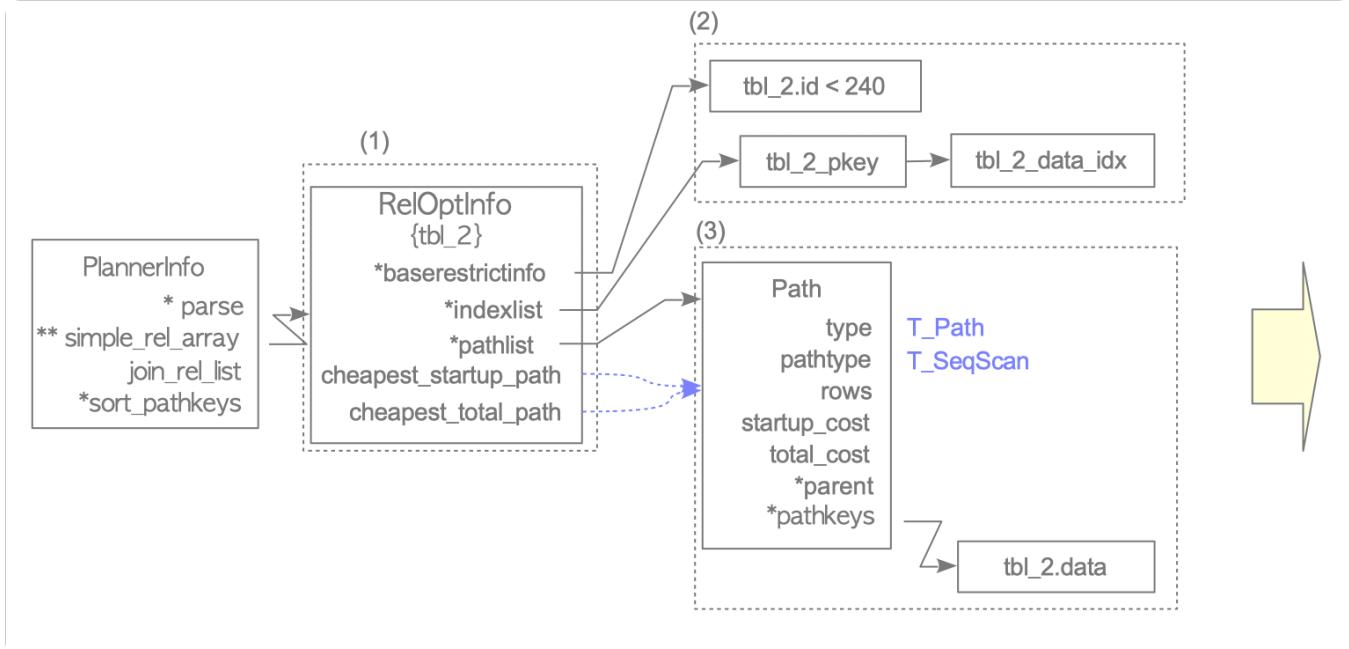
```
testdb=# \d tbl_2
      Table "public.tbl_2"
 Column | Type   | Modifiers
-----+-----+
 id    | integer | not null
 data  | integer |
Indexes:
 "tbl_2_pkey" PRIMARY KEY, btree (id)
 "tbl_2_data_idx" btree (data)
```

```
testdb=# SELECT * FROM tbl_2 WHERE id < 240;
```

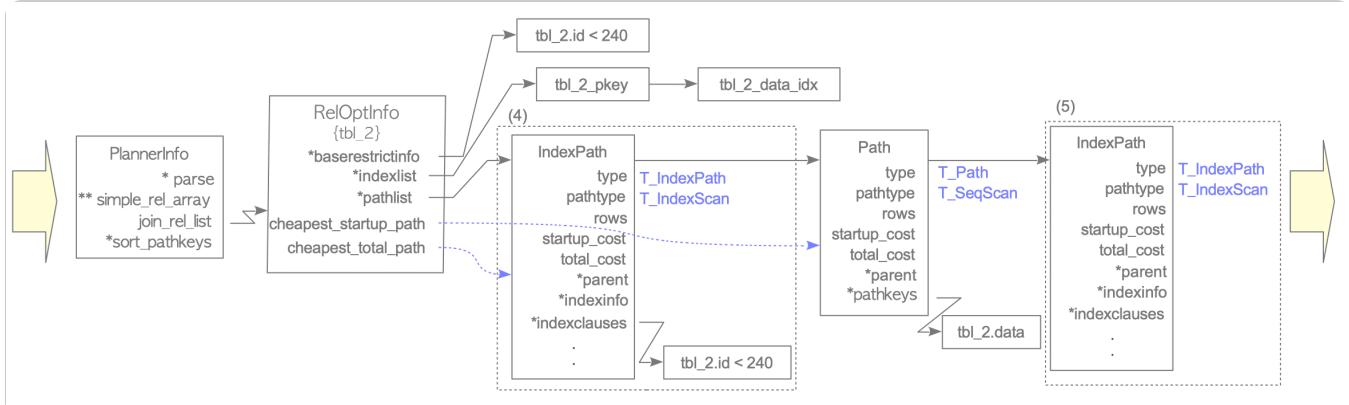
Figures 3.12 to 3.14 depict how the planner performs in this example.

- (1) Create a RelOptInfo structure.
  - (2) Add the WHERE clause to the baserestrictinfo, and add the indexes of the target table to the indexlist.
- In this example, a WHERE clause 'id < 240' is added to the baserestrictinfo, and two indexes, *tbl\_2\_pkey* and *tbl\_2\_data\_idx*, are added to the indexlist of the RelOptInfo.
- (3) Create a path, estimate the cost of the sequential scan, and add the path to the pathlist of the RelOptInfo.

**Fig. 3.12. How to get the cheapest path of Example 2.**



**Fig. 3.13. How to get the cheapest path of Example 2 (continued from Fig. 3.12).**



- (4) Create an `IndexPath`, estimate the cost of the index scan, and add the `IndexPath` to the `pathlist` of the `RelOptInfo` using the `add_path()` function.

In this example, as there are two indexes, *tbl\_2\_pkey* and *tbl\_2\_data\_idx*, these indexes are processed in order. *tbl\_2\_pkey* is processed first.

An `IndexPath` is created for *tbl\_2\_pkey*, and both the start-up and the total costs are estimated. In this example, *tbl\_2\_pkey* is the index related to the column 'id', and the WHERE clause contains the column 'id'; therefore, the WHERE clause is stored in the `indexclauses` of the `IndexPath`.

Note that when adding access paths to the `pathlist`, the `add_path()` function adds paths in the sort order of the total cost. In this example, the total cost of this index scan is smaller than the sequential total cost; thus, this index path is inserted before the sequential scan path.

```

typedef struct IndexPath
{
    Path          path;
    IndexOptInfo *indexinfo;
    List          *indexclauses;
    List          *indexorderbys;
    List          *indexorderbycols;
    ScanDirection indexscandir;
    Cost          indextotalcost;
    Selectivity   indexselectivity;
} IndexPath;

/*
 * IndexOptInfo
 *             Per-index information for planning/optimization
 *
 *             indexkeys[], indexcollations[] each have ncolumns entries.
 *             opfamily[], and opcintype[]      each have nkeycolumns entries. Th
ey do
 *
 *             sortopfamily[], reverse_sort[], and nulls_first[] have
n.
 *             nkeycolumns entries, if the index is ordered; but if it is unorde
red,
 *
 *             those pointers are NULL.
 *
 *             Zeroes in the indexkeys[] array indicate index columns that are
he
 *             expressions; there is one element in indexexprs for each such colum
ward
 *
 *             For an ordered index, reverse_sort[] and nulls_first[] describe t
he
 *             sort ordering of a forward indexscan; we can also consider a back
ward
 *
 *             indexscan, which will generate the reverse ordering.
 *
 *             The indexexprs and indpred expressions have been run through
n.
 *             prepqual.c and eval_const_expressions() for ease of matching to
 *             WHERE clauses. indpred is in implicit-AND form.
 *
 *             indexlist is a TargetEntry list representing the index columns.
 *             It provides an equivalent base-relation Var for each simple colum
n,
 *
 *             and links to the matching indexexprs element for each expression co
lumn.
 *
 *             While most of these fields are filled when the IndexOptInfo is cr
eated
 *
 *             (by plancat.c), indrestrictinfo and predOK are set later, in
check_index_predicates().
 */

#ifndef HAVE_INDEXOPTINFO_TYPEDEF
typedef struct IndexOptInfo IndexOptInfo;
#define HAVE_INDEXOPTINFO_TYPEDEF 1
#endif

struct IndexOptInfo
{
    pg_node_attr(no_copy_equal, no_read, no_query_jumble)
    NodeTag           type;

```

```

/* OID of the index relation */
Oid indexoid;
/* tablespace of index (not table) */
Oid reltablespace;
/* back-link to index's table; don't print, else infinite recursion */
RelOptInfo *rel pg_node_attr(read_write_ignore);

/*
 * index-size statistics (from pg_class and elsewhere)
 */
/* number of disk pages in index */
BlockNumber pages;
/* number of index tuples in index */
Cardinality tuples;
/* index tree height, or -1 if unknown */
int tree_height;

/*
 * index descriptor information
 */
/* number of columns in index */
int ncolumns;
/* number of key columns in index */
int nkeycolumns;

/*
 * table column numbers of index's columns (both key and included
 * columns), or 0 for expression columns
 */
int *indexkeys pg_node_attr(array_size(ncolumns));
/* OIDs of collations of index columns */
Oid *indexcollations pg_node_attr(array_size(nkeycolumn
s));
/* OIDs of operator families for columns */
Oid *opfamily pg_node_attr(array_size(nkeycolumns));
/* OIDs of opclass declared input data types */
Oid *opcintype pg_node_attr(array_size(nkeycolumns));
/* OIDs of btree opfamilies, if orderable. NULL if partitioned index */
Oid *sortopfamily pg_node_attr(array_size(nkeycolumns));
/* is sort order descending? or NULL if partitioned index */
bool *reverse_sort pg_node_attr(array_size(nkeycolumns));
/* do NULLs come first in the sort order? or NULL if partitioned index */
bool *nulls_first pg_node_attr(array_size(nkeycolumns));
/* opclass-specific options for columns */
bytea **opclassoptions pg_node_attr(read_write_ignore);
/* which index cols can be returned in an index-only scan? */
bool *canreturn pg_node_attr(array_size(ncolumns));
/* OID of the access method (in pg_am) */
Oid relam;

/*
 * expressions for non-simple index columns; redundant to print since we
 * print indextlist
 */
List *indexexprs pg_node_attr(read_write_ignore);
/* predicate if a partial index, else NIL */
List *indpred;

/* targetlist representing index columns */
List *indextlist;

```

```

/*
 * parent relation's baserestrictinfo list, less any conditions implied by
 * the index's predicate (unless it's a target rel, see comments in
 * check_index_predicates())
 */
List *indrestrictinfo;

/* true if index predicate matches query */
bool predOK;
/* true if a unique index */
bool unique;
/* is uniqueness enforced immediately? */
bool immediate;
/* true if index doesn't really exist */
bool hypothetical;

/*
 * Remaining fields are copied from the index AM's API struct
 * (IndexAmRoutine). These fields are not set for partitioned indexes.
 */
bool amcanorderbyop;
bool amoptionalkey;
bool amsearcharray;
bool amsearchnulls;
/* does AM have amgettuple interface? */
bool amhasgettuple;
/* does AM have amgetbitmap interface? */
bool amhasgetitemp;
bool amcanparallel;
/* does AM have ammarkpos interface? */
bool amcanmarkpos;
/* AM's cost estimator */
/* Rather than include amapi.h here, we declare amcostestimate like this
 */
void (*amcostestimate) () pg_node_attr(read_write_ignore);
};

}

```

(5) Create another IndexPath, estimate the cost of other index scans, and add the index path to the pathlist of the RelOptInfo.

Next, an IndexPath is created for *tbl\_2\_data\_idx*, the costs are estimated, and this IndexPath is added to the pathlist. In this example, there is no WHERE clause related to the *tbl\_2\_data\_idx* index; therefore, the index clauses are NULL.



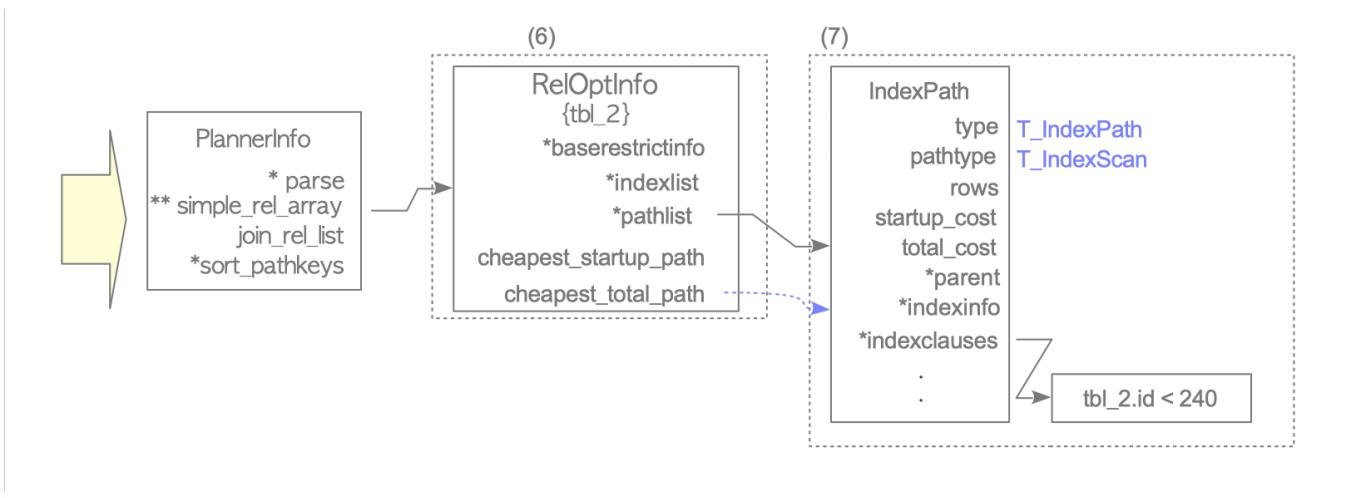
Note that the add\_path() function does not always add the path. The details are omitted because of the complicated nature of this operation. For details, refer to the comment of the add\_path() function.

(6) Create a new RelOptInfo structure.

(7) Add the cheapest path to the pathlist of the new RelOptInfo.

In this example, the cheapest path is the index path using the index *tbl\_2\_pkey*; thus, its path is added to the pathlist of the new RelOptInfo.

**Fig. 3.14. How to get the cheapest path of Example 2 (continued from Fig. 3.13).**



### 3.3.3. Creating a Plan Tree

At the last stage, the planner generates a plan tree from the cheapest path.

The root of the plan tree is a `PlannedStmt` structure defined in `plannodes.h`. It contains nineteen fields, but here are four representative fields:

- **commandType** stores a type of operation, such as SELECT, UPDATE or INSERT.
- **rtable** stores rangeTable entries.
- **relationOids** stores oids of the related tables for this query.
- **plantree** stores a plan tree that is composed of plan nodes, where each node corresponds to a specific operation, such as sequential scan, sort and index scan.

```

typedef struct PlannedStmt
{
    pg_node_attr(no_equal, no_query_jumble)

    NodeTag           type;

    CmdType           commandType; /* select|insert|update|delete|merge|
utility */

    uint64            queryId;      /* query identifier (copied from Quer-
y) */

    bool              hasReturning; /* is it insert|update|delete RETURNING? */

    bool              hasModifyingCTE; /* has insert|update|delete in WITH? */

    bool              canSetTag;     /* do I set the command result tag? */

    bool              transientPlan; /* redo plan when TransactionXmin changes? */

    bool              dependsOnRole; /* is plan specific to current role? */

    bool              parallelModeNeeded; /* parallel mode required to execute? */

    int               jitFlags;     /* which forms of JIT should
be performed */
  
```

```

    struct Plan *planTree;           /* tree of Plan nodes */

    List      *rtable;              /* list of RangeTblEntry nodes */

    List      *permInfos;           /* list of RTEPermissionInfo nodes for rtable
                                     * entries needing on
                                     */

/* rtable indexes of target relations for INSERT/UPDATE/DELETE/MERGE */
List      *resultRelations;      /* integer list of RT indexes, or NIL */

List      *appendRelations;      /* list of AppendRelInfo nodes */

List      *subplans;              /* Plan trees for SubPlan expressions; note
                                     * that some could be
                                     */

NULL */

    Bitmapset *rewindPlanIDs;       /* indices of subplans that require REWIND */

    List      *rowMarks;             /* a list of PlanRowMark's */

    List      *relationOids;         /* OIDs of relations the plan depends on */

    List      *invalidItems;          /* other dependencies, as PlanInvalItems */

    List      *paramExecTypes;        /* type OIDs for PARAM_EXEC Params */

    Node     *utilityStmt;           /* non-null if this is utility stmt */

    /* statement location in source string (copied from Query) */
    int      stmt_location;          /* start location, or -1 if unknown
                                     */
/*
    int                  stmt_len;            /* length in bytes; 0 means
                                             "rest of string" */
} PlannedStmt;

```

As mentioned above, a plan tree is composed of various plan nodes. The `PlanNode` structure is the base node, and other nodes always contain it. For example, `SeqScanNode`, which is for sequential scanning, is composed of a `PlanNode` and an integer variable '`scanrelid`'. A `PlanNode` contains fourteen fields. The following are seven representative fields.

- **start-up cost** and **total\_cost** are the estimated costs of the operation corresponding to this node.
- **rows** is the number of rows to be scanned, which is estimated by the planner.
- **targetlist** stores the target list items contained in the query tree.
- **qual** is a list that stores qual conditions.
- **lefttree** and **righttree** are the nodes for adding the children nodes.

```

/*
 *          Plan node
 *
 * All plan nodes "derive" from the Plan structure by having the
 * Plan structure as the first field. This ensures that everything works
 * when nodes are cast to Plan's. (node pointers are frequently cast to Plan)
 * when passed around generically in the executor)
 *
 * We never actually instantiate any Plan nodes; this is just the common
 * abstract superclass for all Plan-type nodes.
*/

```

```

*/
typedef struct Plan
{
    pg_node_attr(abstract, no_equal, no_query_jumble)

    NodeTag           type;

    /*
     * estimated execution costs for plan (see costsize.c for more info)
     */
    Cost             startup_cost; /* cost expended before fetching any tuples
*/
    Cost             total_cost;      /* total cost (assuming all tuples fetched) */

    /*
     * planner's estimate of result size of this plan step
     */
    Cardinality plan_rows;          /* number of rows plan is expected to emit */
    int              plan_width;     /* average row width in bytes
*/

    /*
     * information needed for parallel query
     */
    bool            parallel_aware; /* engage parallel-aware logic? */
    bool            parallel_safe;  /* OK to use as part of parallel plan? */

    /*
     * information needed for asynchronous execution
     */
    bool            async_capable; /* engage asynchronous-capable logic? */

    /*
     * Common structural data for all Plan types.
     */
    int              plan_node_id;   /* unique across entire final plan tree */
    List            *targetlist;     /* target list to be computed at this node */
    List            *qual;           /* implicitly-ANDed qual conditions
*/
    struct Plan *lefttree;        /* input plan tree(s) */
    struct Plan *righttree;
    List            *initPlan;       /* Init Plan nodes (un-correlated expr
                                     * subselects) */

    /*
     * Information for management of parameter-change-driven rescanning
     *
     * extParam includes the paramIDs of all external PARAM_EXEC params
     * affecting this plan node or its children.  setParam params from the
     * node's initPlans are not included, but their extParams are.
     *
     * allParam includes all the extParam paramIDs, plus the IDs of local
     * params that affect the node (i.e., the setParams of its initplans).
     * These are all the PARAM_EXEC params that affect this node.
     */
    Bitmapset      *extParam;
    Bitmapset      *allParam;
} Plan;

```

```

/*
 * =====
 * Scan nodes
 *
 * Scan is an abstract type that all relation scan plan types inherit from.
 * =====
 */
typedef struct Scan
{
    pg_node_attr(abstract)

    Plan          plan;
    Index         scanrelid;           /* relid is index into the range table
e */
} Scan;

/*
 * -----
 *           sequential scan node
 * -----
 */
typedef struct SeqScan
{
    Scan          scan;
} SeqScan;

```

In the following, two plan trees, which will be generated from the cheapest paths shown in the examples in the previous subsection, are described.

### 3.3.3.1. Example 1

The first example is the plan tree of the example in Section 3.3.2.1. The cheapest path shown in Figure 3.11 is a tree composed of a sort path and a sequential scan path. The root path is the sort path, and the child path is the sequential scan path. Although detailed explanations are omitted, it will be easy to understand that the plan tree can be almost trivially generated from the cheapest path. In this example, a SortNode is added to the plantree of the PlannedStmt structure, and a SeqScanNode is added to the lefttree of the SortNode. See Fig. 3.15(a).

```

/*
 * -----
 *           sort node
 * -----
 */
typedef struct Sort
{
    Plan          plan;

    /* number of sort-key columns */
    int           numCols;

    /* their indexes in the target list */
    AttrNumber *sortColIdx pg_node_attr(array_size(numCols));

    /* OIDs of operators to sort them by */
    Oid           *sortOperators pg_node_attr(array_size(numCols));

    /* OIDs of collations */
    Oid           *collations pg_node_attr(array_size(numCols));

    /* NULLS FIRST/LAST directions */

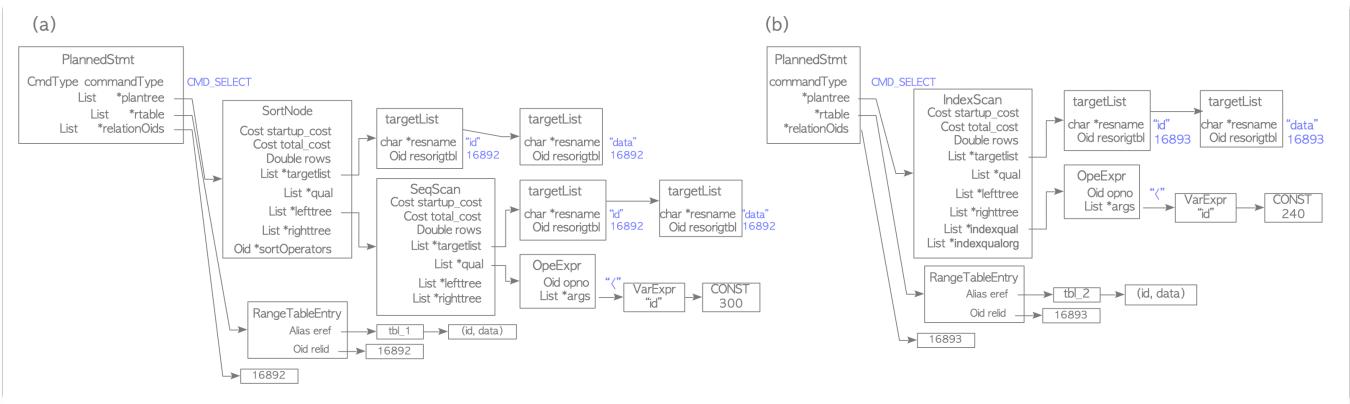
```

```

    bool *nullsFirst pg_node_attr(array_size(numCols));
} Sort;

```

**Fig. 3.15. Examples of plan trees.**



In the SortNode, the lefttree points to the SeqScanNode.

In the SeqScanNode, the qual holds the WHERE clause 'id < 300'.

### 3.3.3.2. Example 2

The second example is the plan tree of the example in Section 3.3.2.2. The cheapest path shown in Fig. 3.14 is the index scan path, so the plan tree is composed of an **IndexScanNode** structure alone. See Fig. 3.15(b).

```

/*
 *          index scan node
 *
 * indexqualorig is an implicitly-ANDed list of index qual expressions, each
 * in the same form it appeared in the query WHERE condition. Each should
 * be of the form (indexkey OP comparisonval) or (comparisonval OP indexkey).
 * The indexkey is a Var or expression referencing column(s) of the index's
 * base table. The comparisonval might be any expression, but it won't use
 * any columns of the base table. The expressions are ordered by index
 * column position (but items referencing the same index column can appear
 * in any order). indexqualorig is used at runtime only if we have to recheck
 * a lossy indexqual.
 *
 * indexqual has the same form, but the expressions have been commuted if
 * necessary to put the indexkeys on the left, and the indexkeys are replaced
 * by Var nodes identifying the index columns (their varno is INDEX_VAR and
 * their varattno is the index column number).
 *
 * indexorderbyorig is similarly the original form of any ORDER BY expressions
 * that are being implemented by the index, while indexorderby is modified to
 * have index column Vars on the left-hand side. Here, multiple expressions
 * must appear in exactly the ORDER BY order, and this is not necessarily the
 * index column order. Only the expressions are provided, not the auxiliary
 * sort-order information from the ORDER BY SortGroupClauses; it's assumed
 * that the sort ordering is fully determinable from the top-level operators.
 * indexorderbyorig is used at runtime to recheck the ordering, if the index
 * cannot calculate an accurate ordering. It is also needed for EXPLAIN.
 *
 * indexorderbyops is a list of the OIDs of the operators used to sort the
 * ORDER BY expressions. This is used together with indexorderbyorig to
 * recheck ordering at run time. (Note that indexorderby, indexorderbyorig,
 * and indexorderbyops are used for amcanorderbyop cases, not amcanorder.)

```

```

/*
 * indexorderdir specifies the scan ordering, for indexscans on amcanorder
 * indexes (for other indexes it should be "don't care").
 */
typedef struct Scan
{
    pg_node_attr(abstract)

    Plan          plan;
    Index         scanrelid;           /* relid is index into the range tabl
e */
} Scan;

typedef struct IndexScan
{
    Scan          scan;
    Oid           indexid;           /* OID of index to scan */
    List          *indexqual;        /* list of index quals (usually OpExprs) */
    List          *indexqualorig;   /* the same in original form */
    List          *indexorderby;     /* list of index ORDER BY exprs */
    List          *indexorderbyorig; /* the same in original form */
    List          *indexorderbyops;  /* OIDs of sort ops for ORDER BY exprs */
    ScanDirection indexorderdir;   /* forward or backward or don't care */
} IndexScan;

```

In this example, the WHERE clause 'id < 240' is an access predicate, so it is stored in the indexqual of the IndexScanNode.

## 3.4. How the Executor Performs

In single-table queries, the executor takes the plan nodes in an order from the end of the plan tree to the root and then invokes the functions that perform the processing of the corresponding nodes.

Each plan node has functions that are meant for executing the respective operation. These functions are located in the `src/backend/executor/` directory. For example, the functions for executing the sequential scan (`ScanScan`) are defined in `nodeSeqscan.c`; the functions for executing the index scan (`IndexScanNode`) are defined in `nodeIndexscan.c`; the functions for sorting `SortNode` are defined in `nodeSort.c`, and so on.

Of course, the best way to understand how the executor performs is to read the output of the EXPLAIN command. PostgreSQL's EXPLAIN shows the plan tree almost as it is. It will be explained using Example 1 in Section 3.3.3.

```

1. testdb=# EXPLAIN SELECT * FROM tbl_1 WHERE id < 300 ORDER BY data;
2.                                     QUERY PLAN
3. -----
4. Sort  (cost=182.34..183.09 rows=300 width=8)
5.   Sort Key: data
6.   -> Seq Scan on tbl_1  (cost=0.00..170.00 rows=300 width=8)
7.       Filter: (id < 300)
8.   (4 rows)

```

Let's explore how the executor performs. Read the result of the EXPLAIN command from the bottom line to the top line.

**Line 6:** At first, the executor carries out a sequential scan operation using the functions defined in nodeSeqscan.c.

**Line 4:** Next, the executor sorts the result of the sequential scan using the functions defined in nodeSort.c.

## ❶ Temporary Files

Although the executor uses the work\_men and temp\_buffers, which are allocated in the memory, for query processing, it uses temporary files if the processing cannot be performed within the memory alone.

Using the ANALYZE option, the EXPLAIN command actually executes the query and displays the true row counts, true run time, and the actual memory usage. A specific example is shown below:

```
1. testdb=# EXPLAIN ANALYZE SELECT id, data FROM tbl_25m ORDER BY id;
2.                                                 QUERY PLAN
3.
4. Sort  (cost=3944070.01..3945895.01 rows=730000 width=4104) (actual time=885.648..1033.746 rows=73
   0000 loops=1)
5.   Sort Key: id
6.   Sort Method: external sort Disk: 10000kB
7.     -> Seq Scan on tbl_25m  (cost=0.00..10531.00 rows=730000 width=4104) (actual time=0.024..102.5
   48 rows=730000 loops=1)
8. Planning time: 1.548 ms
9. Execution time: 1109.571 ms
10. (6 rows)
```

In Line 6, the EXPLAIN command shows that the executor has used a temporary file whose size is 10000kB.

Temporary files are created in the base/pg\_tmp subdirectory temporarily, and the naming method is shown follows:

```
{"pgsql_tmp"} + {PID of the postgres process which creates the file} . {sequential number from 0}
```

For example, the temporary file 'pgsql\_tmp8903.5' is the 6th temporary file created by the postgres process with the pid of 8903.

```
$ ls -la /usr/local/pgsql/data/basepgsql_tmp*
-rw----- 1 postgres postgres 10240000 12 4 14:18 pgsql_tmp8903.5
```

## 3.5. Join Operations

Go to Section 3.5.

[← Back to Part 1](#)      [Go to Part 3 →](#)

# Chapter 3

## Query Processing (Part 3)

---

[← Back to Part 1, Part 2](#)

### 3.5. Join Operations

PostgreSQL supports three join operations: nested loop join, merge join and hash join. The nested loop join and the merge join in PostgreSQL have several variations.

In the following, we assume that the reader is familiar with the basic behavior of these three joins. If you are unfamiliar with these terms, see [1, 2]. However, as there is not much explanation on the hybrid hash join with skew supported by PostgreSQL, it will be explained in more detail here.

Note that the three join methods supported by PostgreSQL can perform all join operations, not only INNER JOIN, but also LEFT/RIGHT OUTER JOIN, FULL OUTER JOIN, and so on. However, for simplicity, we focus on the NATURAL INNER JOIN in this chapter.

#### 3.5.1. Nested Loop Join

The nested loop join is the most fundamental join operation, and it can be used in all join conditions. PostgreSQL supports the nested loop join and five variations of it.

##### 3.5.1.1. Nested Loop Join

The nested loop join does not need any start-up operation, so the start-up cost is 0:

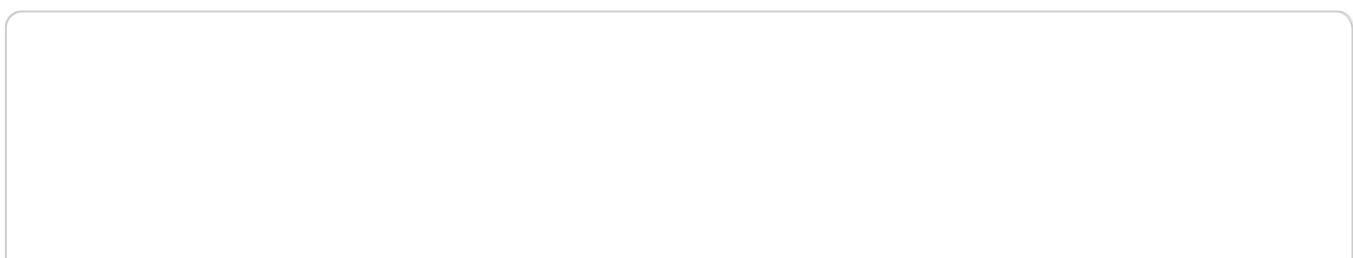
$$\text{'start-up cost'} = 0.$$

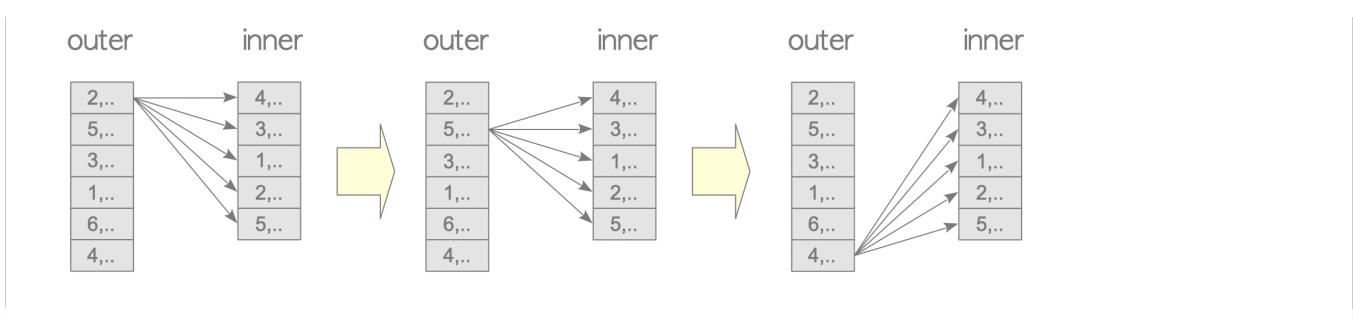
The run cost of the nested loop join is proportional to the product of the sizes of the outer and inner tables. In other words, the 'run cost' is  $O(N_{outer} \times N_{inner})$ , where  $N_{outer}$  and  $N_{inner}$  are the numbers of tuples of the outer table and the inner table, respectively. More precisely, the run cost is defined by the following equation:

$$\text{'run cost'} = (\text{cpu\_operator\_cost} + \text{cpu\_tuple\_cost}) \times N_{outer} \times N_{inner} + C_{inner} \times N_{outer} + C_{outer}$$

where  $C_{outer}$  and  $C_{inner}$  are the scanning costs of the outer table and the inner table, respectively.

**Fig. 3.16. Nested loop join.**





The cost of the nested loop join is always estimated, but this join operation is rarely used because more efficient variations, which are described in the following, are usually used.

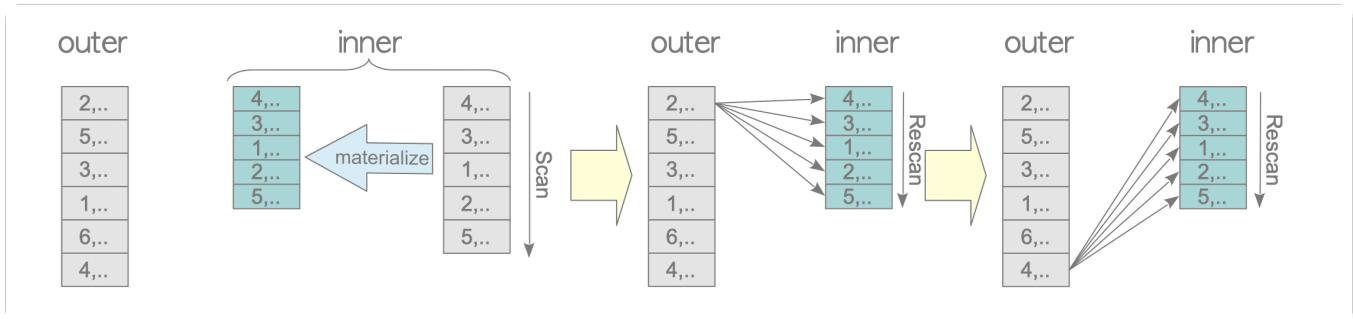
### **3.5.1.2. Materialized Nested Loop Join**

The nested loop join described above has to scan all the tuples of the inner table whenever each tuple of the outer table is read. Since scanning the entire inner table for each outer table tuple is a costly process, PostgreSQL supports the *materialized nested loop join* to reduce the total scanning cost of the inner table.

Before running a nested loop join, the executor writes the inner table tuples to the `work_mem` or a temporary file by scanning the inner table once using the *temporary tuple storage* module described in ❶ below. This has the potential to process the inner table tuples more efficiently than using the buffer manager, especially if at least all the tuples are written to `work_mem`.

Figure 3.17 illustrates how the materialized nested loop join performs. Scanning materialized tuples is internally called **rescan**.

**Fig. 3.17.** Materialized nested loop join.



## Temporary Tuple Storage

PostgreSQL internally provides a temporary tuple storage module for materializing tables, creating batches in hybrid hash join and so on. This module is composed of the functions defined in `tuplestore.c`, and they store and read a sequence of tuples to/from `work_mem` or temporary files. The decision of whether to use the `work_mem` or temporary files depends on the total size of the tuples to be stored.

We will explore how the executor processes the plan tree of the materialized nested loop join and how the cost is estimated using the specific example shown below.

```

4. Nested Loop  (cost=0.00..750230.50 rows=5000 width=16)
5.   Join Filter: (a.id = b.id)
6.     -> Seq Scan on tbl_a a  (cost=0.00..145.00 rows=10000 width=8)
7.     -> Materialize  (cost=0.00..98.00 rows=5000 width=8)
8.       -> Seq Scan on tbl_b b  (cost=0.00..73.00 rows=5000 width=8)
9.   (5 rows)

```

First, the operation of the executor is shown. The executor processes the displayed plan nodes as follows:

**Line 7:** The executor materializes the inner table `tbl_b` by sequential scanning (Line 8).

**Line 4:** The executor carries out the nested loop join operation; the outer table is `tbl_a` and the inner one is the materialized `tbl_b`.

In what follows, the costs of the 'Materialize' (Line 7) and 'Nested Loop' (Line 4) operations are estimated. Assume that the materialized inner tuples are stored in the `work_mem`.

### Materialize:

There is no cost to start up, so the start-up cost is 0.

$$\text{'start-up cost'} = 0$$

The run cost is defined by the following equation:

$$\text{'run cost'} = 2 \times \text{cpu\_operator\_cost} \times N_{inner}$$

Therefore,

$$\text{'run cost'} = 2 \times 0.0025 \times 5000 = 25.0.$$

In addition, the total cost is the sum of the startup cost, the total cost of the sequential scan, and the run cost:

$$\text{'total cost'} = (\text{'start-up cost'} + \text{'total cost of seq scan'}) + \text{'run cost'}$$

Therefore,

$$\text{'total cost'} = (0.0 + 73.0) + 25.0 = 98.0.$$

### (Materialized) Nested Loop:

There is no cost to start up, so the start-up cost is 0.

$$\text{'start-up cost'} = 0$$

Before estimating the run cost, we consider the *rescan cost*. This cost is defined by the following equation:

$$\text{'rescan cost'} = \text{cpu\_operator\_cost} \times N_{inner}$$

In this case,

$$\text{'rescan cost'} = (0.0025) \times 5000 = 12.5.$$

The run cost is defined by the following equation:

$$\begin{aligned}
 \text{'run cost'} &= (\text{cpu\_operator\_cost} + \text{cpu\_tuple\_cost}) \times N_{\text{inner}} \times N_{\text{outer}} \\
 &+ \text{'rescan cost'} \times (N_{\text{outer}} - 1) + C_{\text{outer}, \text{seqscan}}^{\text{total}} + C_{\text{materialize}}^{\text{total}},
 \end{aligned}$$

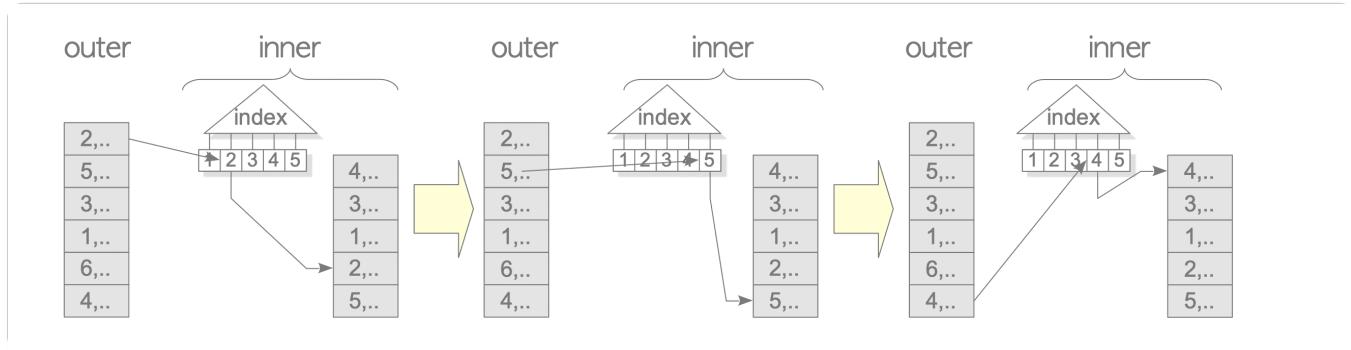
where  $C_{\text{outer}, \text{seqscan}}^{\text{total}}$  is the total scan cost of the outer table and  $C_{\text{materialize}}^{\text{total}}$  is the total cost of the materialized. Therefore,

$$\text{'run cost'} = (0.0025 + 0.01) \times 5000 \times 10000 + 12.5 \times (10000 - 1) + 145.0 + 98.0 = 750230.5.$$

### 3.5.1.3. Indexed Nested Loop Join

If there is an index on the inner table that can be used to look up the tuples satisfying the join condition for each tuple of the outer table, the planner will consider using this index for directly searching the inner table tuples instead of sequential scanning. This variation is called **indexed nested loop join**; see Fig. 3.18. Despite the name, this algorithm can process all the tuples of the outer table in a single loop, so it can perform the join operation efficiently.

**Fig. 3.18. Indexed nested loop join.**



A specific example of the indexed nested loop join is shown below.

```

1. testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_b AS b WHERE c.id = b.id;
2.                                     QUERY PLAN
3.
4. Nested Loop  (cost=0.29..1935.50 rows=5000 width=16)
5.   -> Seq Scan on tbl_b b (cost=0.00..73.00 rows=5000 width=8)
6.   -> Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..0.36 rows=1 width=8)
7.         Index Cond: (id = b.id)
8. (4 rows)

```

In Line 6, the cost of accessing a tuple of the inner table is displayed. This is the cost of looking up the inner table if the tuple satisfies the index condition ( $\text{id} = \text{b.id}$ ), which is shown in Line 7.

In the index condition ( $\text{id} = \text{b.id}$ ) in Line 7, ' $\text{b.id}$ ' is the value of the outer table's attribute used in the join condition. Whenever a tuple of the outer table is retrieved by sequential scanning, the index scan path in Line 6 looks up the inner tuples to be joined. In other words, whenever the outer table is passed as a parameter, this index scan path looks up the inner tuples that satisfy the join condition. Such an index path is called a **parameterized (index) path**. Details are described in README.

The start-up cost of this nested loop join is equal to the cost of the index scan in Line 6; thus,

$$\text{'start-up cost'} = 0.285.$$

The total cost of the indexed nested loop join is defined by the following equation:

$$\text{'total cost'} = (\text{cpu\_tuple\_cost} + C_{\text{inner,parameterized}}^{\text{total}}) \times N_{\text{outer}} + C_{\text{outer,seqscan}}^{\text{run}},$$

where  $C_{\text{inner,parameterized}}^{\text{total}}$  is the total cost of the parameterized inner index scan.

In this case,

$$\text{'total cost'} = (0.01 + 0.3625) \times 5000 + 73.0 = 1935.5,$$

and the run cost is

$$\text{'run cost'} = 1935.5 - 0.285 = 1935.215.$$

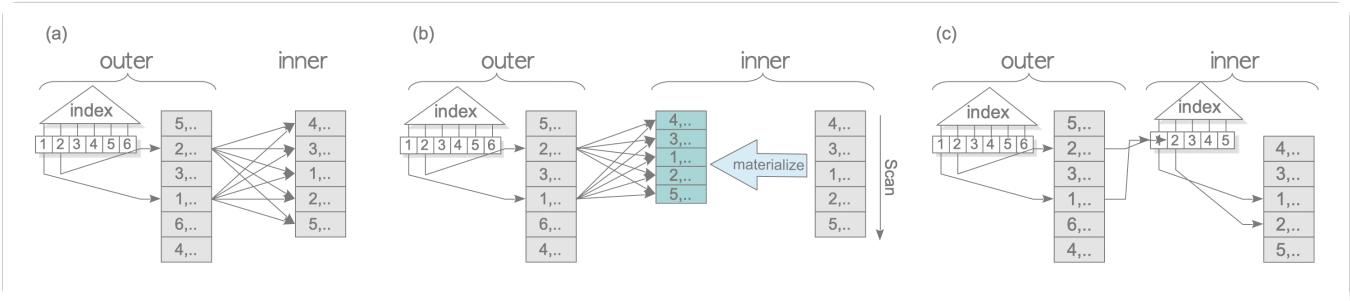
As shown above, the total cost of the indexed nested loop is  $O(N_{\text{outer}})$ .

### 3.5.1.4. Other Variations

If there is an index of the outer table and its attributes are involved in the join condition, it can be used for index scanning instead of the sequential scan of the outer table. In particular, if there is an index whose attribute can be used as an access predicate in the WHERE clause, the search range of the outer table is narrowed. This can drastically reduce the cost of the nested loop join.

PostgreSQL supports three variations of the nested loop join with an outer index scan. See Fig. 3.19.

**Fig. 3.19. The three variations of the nested loop join with an outer index scan.**



The results of these joins' EXPLAIN are shown here.

### 3.5.2. Merge Join

Unlike the nested loop join, the merge join can only be used in natural joins and equi-joins.

The cost of the merge join is estimated by the `initial_cost_mergejoin()` and `final_cost_mergejoin()` functions.

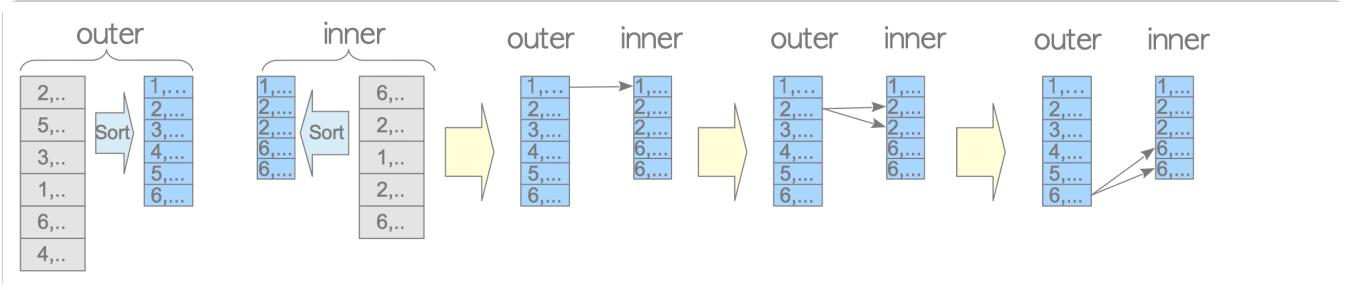
The exact cost estimation is complicated, so it is omitted here. Instead, we will only the runtime order of the merge join algorithm. The start-up cost of the merge join is the sum of sorting costs of both inner and outer tables. This means that the start-up cost is  $O(N_{\text{outer}} \log_2(N_{\text{outer}}) + N_{\text{inner}} \log_2(N_{\text{inner}}))$ , where  $N_{\text{outer}}$  and  $N_{\text{inner}}$  are the number of tuples of the outer and inner tables, respectively. The run cost is  $O(N_{\text{outer}} + N_{\text{inner}})$ .

Similar to the nested loop join, the merge join in PostgreSQL has four variations.

#### 3.5.2.1. Merge Join

Figure 3.20 shows a conceptual illustration of a merge join.

**Fig. 3.20. Merge join.**



If all tuples can be stored in memory, the sorting operations will be able to be carried out in memory itself. Otherwise, temporary files will be used.

A specific example of the EXPLAIN command's result of the merge join is shown below.

```

1. testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND b.id
   < 1000;
2.                                     QUERY PLAN
3.
4. Merge Join  (cost=944.71..984.71 rows=1000 width=16)
5.   Merge Cond: (a.id = b.id)
6.   -> Sort  (cost=809.39..834.39 rows=10000 width=8)
7.     Sort Key: a.id
8.     -> Seq Scan on tbl_a a  (cost=0.00..145.00 rows=10000 width=8)
9.   -> Sort  (cost=135.33..137.83 rows=1000 width=8)
10.    Sort Key: b.id
11.    -> Seq Scan on tbl_b b  (cost=0.00..85.50 rows=1000 width=8)
12.    Filter: (id < 1000)
13.   (9 rows)

```

**Line 9:** The executor sorts the inner table `tbl_b` using sequential scanning (Line 11).

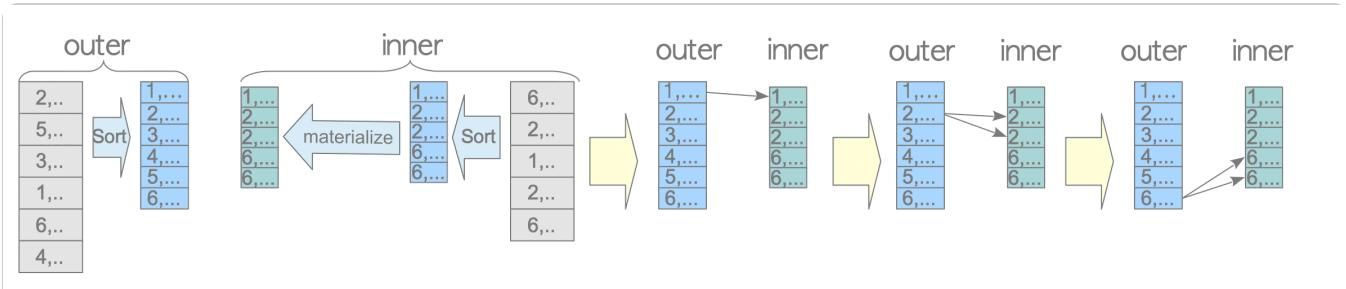
**Line 6:** The executor sorts the outer table `tbl_a` using sequential scanning (Line 8).

**Line 4:** The executor carries out a merge join operation; the outer table is the sorted `tbl_a` and the inner one is the sorted `tbl_b`.

### 3.5.2.2. Materialized Merge Join

Same as in the nested loop join, the merge join also supports the materialized merge join to materialize the inner table to make the inner table scan more efficient.

**Fig. 3.21. Materialized merge join.**



An example of the result of the materialized merge join is shown. It is easy to see that the difference from the merge join result above is Line 9: 'Materialize'.

```

1. testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id;
2.                                     QUERY PLAN
3.
4. --
5. Merge Join  (cost=10466.08..10578.58 rows=5000 width=2064)
6.   Merge Cond: (a.id = b.id)
7.   -> Sort  (cost=6708.39..6733.39 rows=10000 width=1032)
8.     Sort Key: a.id
9.     -> Seq Scan on tbl_a a  (cost=0.00..1529.00 rows=10000 width=1032)
10.    -> Materialize  (cost=3757.69..3782.69 rows=5000 width=1032)
11.      -> Sort  (cost=3757.69..3770.19 rows=5000 width=1032)
12.        Sort Key: b.id
13.        -> Seq Scan on tbl_b b  (cost=0.00..1193.00 rows=5000 width=1032)
2)
(9 rows)

```

**Line 10:** The executor sorts the inner table `tbl_b` using sequential scanning (Line 12).

**Line 9:** The executor materializes the result of the sorted `tbl_b`.

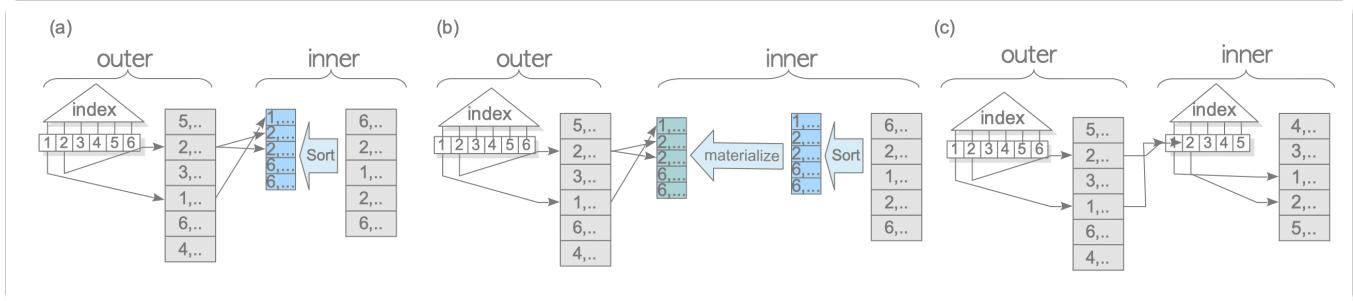
**Line 6:** The executor sorts the outer table `tbl_a` using sequential scanning (Line 8).

**Line 4:** The executor carries out a merge join operation; the outer table is the sorted `tbl_a` and the inner one is the materialized sorted `tbl_b`.

### 3.5.2.3. Other Variations

Similar to the nested loop join, the merge join in PostgreSQL also has variations based on which the index scanning of the outer table can be carried out.

**Fig. 3.22. The three variations of the merge join with an outer index scan.**



The results of these joins' EXPLAIN are shown [here](#).

### 3.5.3. Hash Join

Similar to the merge join, the hash join can be only used in natural joins and equi-joins.

The hash join in PostgreSQL behaves differently depending on the sizes of the tables. If the target table is small enough (more precisely, the size of the inner table is 25% or less of the `work_mem`), it will be a simple two-phase in-memory hash join. Otherwise, the hybrid hash join is used with the skew method.

In this subsection, the execution of both hash joins in PostgreSQL is described.

Discussion of the cost estimation has been omitted because it is complicated. Roughly speaking, the start-up and run costs are  $O(N_{outer} + N_{inner})$  if assuming there is no conflict when searching and

inserting into a hash table.

### 3.5.3.1. In-Memory Hash Join

In this subsection, the in-memory hash join is described.

This in-memory hash join is processed in the `work_mem`, and the hash table area is called a **batch** in PostgreSQL. A batch has hash *slots*, internally called **buckets**, and the number of buckets is determined by the `ExecChooseHashTableSize()` function defined in `nodeHash.c`; the number of buckets is always  $2^n$ , where  $n$  is an integer.

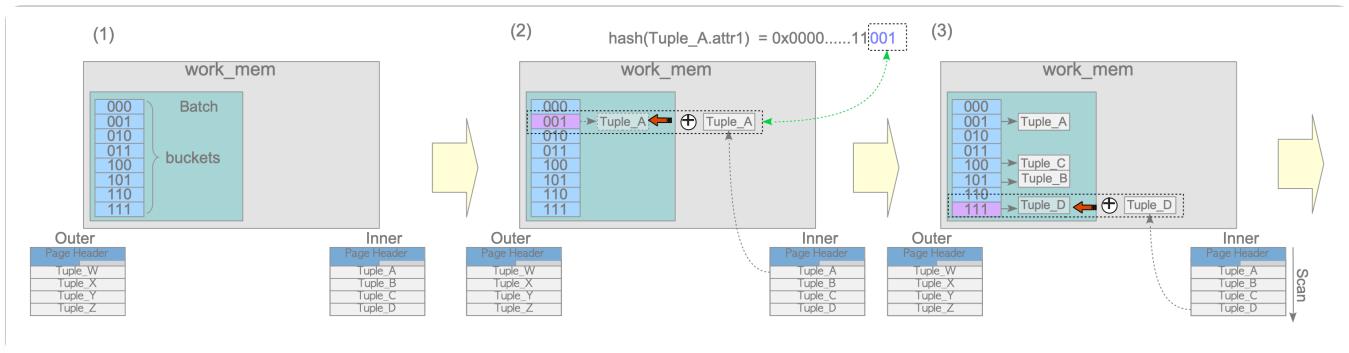
The in-memory hash join has two phases: the **build** and the **probe** phases. In the build phase, all tuples of the inner table are inserted into a batch; in the probe phase, each tuple of the outer table is compared with the inner tuples in the batch and joined if the join condition is satisfied.

A specific example is shown to clearly understand this operation. Assume that the query shown below is executed using a hash join.

```
testdb=# SELECT * FROM tbl_outer AS outer, tbl_inner AS inner WHERE inner.attr1 = outer.attr2;
```

In the following, the operation of a hash join is shown. Refer to Figs. 3.23 and 3.24.

**Fig. 3.23. The build phase in the in-memory hash join.**



(1) Create a batch on `work_mem`.

In this example, the batch has 8 buckets, which means the number of buckets is  $2^3$ .

(2) Insert the first tuple of the inner table into the corresponding bucket of the batch.

The details are as follows:

1. Calculate the hash-key of the first tuple's attribute that is involved in the join condition.

In this example, the hash-key of the attribute 'attr1' of the first tuple is calculated using the built-in hash function because the WHERE clause is 'inner.attr1 = outer.attr2'.

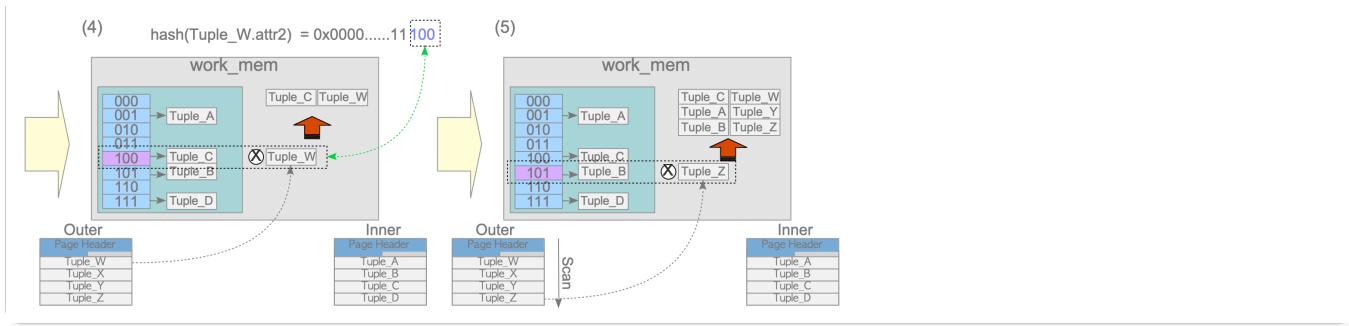
2. Insert the first tuple into the corresponding bucket.

Assume that the hash-key of the first tuple is '0x000...001' by binary notation, which means the last three bits are '001'. In this case, this tuple is inserted into the bucket whose key is '001'.

In this document, this insertion operation to build a batch is represented by this operator:  $\oplus$

(3) Insert the remaining tuples of the inner table.

**Fig. 3.24. The probe phase in the in-memory hash join.**



#### (4) Probe the first tuple of the outer table.

The details are as follows:

1. Calculate the hash key of the first tuple's attribute that is involved in the join condition of the outer table.

In this example, assume that the hash-key of the first tuple's attribute 'attr2' is '0x000...100'; that is, the last three bits are '100'.

2. Compare the first tuple of the outer table with the inner tuples in the batch and join tuples if the join condition is satisfied.

Because the last three bits of the hash-key of the first tuple are '100', the executor retrieves the tuples belonging to the bucket whose key is '100' and compares both values of the respective attributes of the tables specified by the join condition (defined by the WHERE clause).

If the join condition is satisfied, the first tuple of the outer table and the corresponding tuple of the inner table will be joined. Otherwise, the executor does nothing.

In this example, the bucket whose key is '100' has Tuple\_C. If the attr1 of Tuple\_C is equal to the attr2 of the first tuple (Tuple\_W), then Tuple\_C and Tuple\_W will be joined and saved to memory or a temporary file.

In this document, such operation to probe a batch is represented by the operator:  $\otimes$

#### (5) Probe the remaining tuples of the outer table.

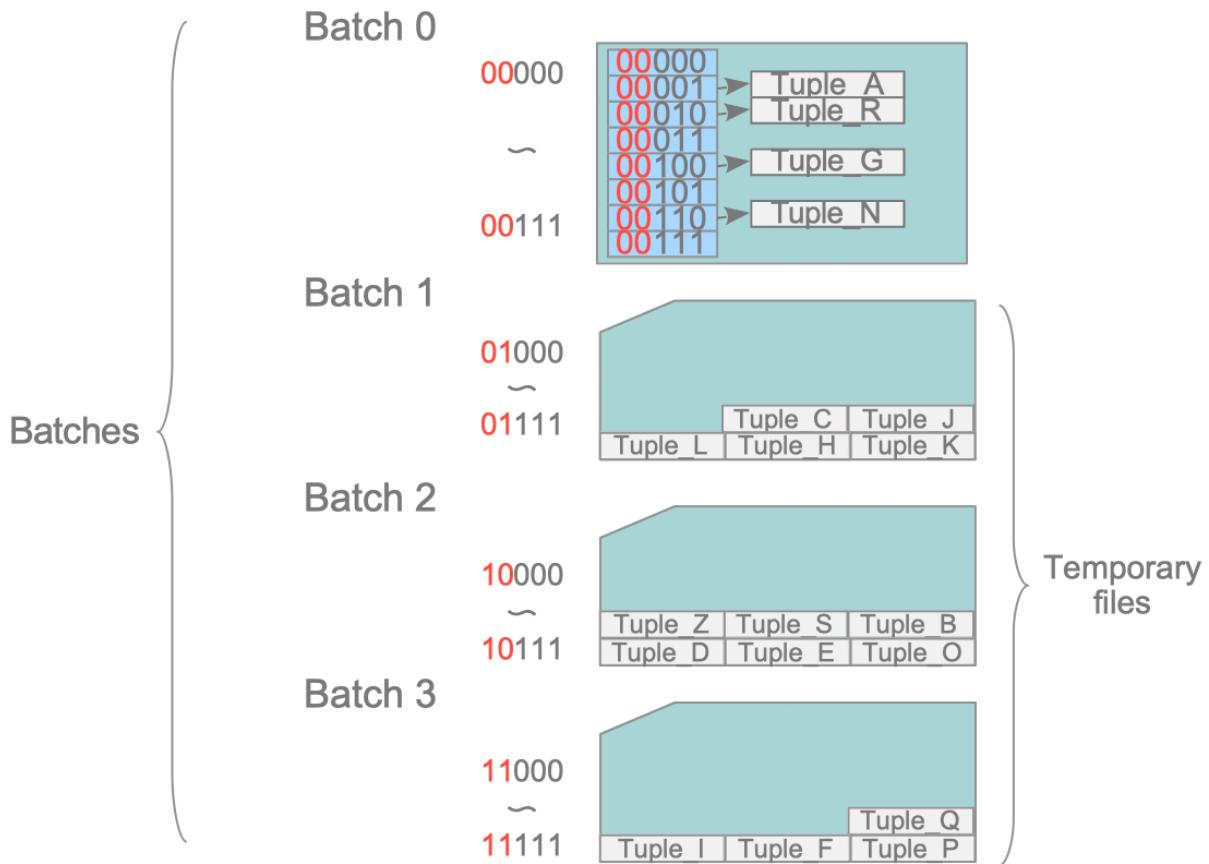
### 3.5.3.2. Hybrid Hash Join with Skew

When the tuples of the inner table cannot be stored in one batch in `work_mem`, PostgreSQL uses the hybrid hash join with the skew algorithm, which is a variation based on the hybrid hash join.

First, the basic concept of the hybrid hash join is described. In the first build and probe phases, PostgreSQL prepares multiple batches. The number of batches is the same as the number of buckets, determined by the `ExecChooseHashTableSize()` function. It is always  $2^m$ , where  $m$  is an integer. At this stage, only one batch is allocated in `work_mem`, and the other batches are created as temporary files. The tuples belonging to these batches are written to the corresponding files and saved using the temporary tuple storage feature.

Figure 3.25 illustrates how tuples are stored in four ( $=2^2$ ) batches. In this case, which batch stores each tuple is determined by the first two bits of the last 5 bits of the tuple's hash-key, because the sizes of the buckets and batches are  $2^3$  and  $2^2$ , respectively. Batch\_0 stores the tuples whose last 5 bits of the hash-key are between '00000' and '00111', Batch\_1 stores the tuples whose last 5 bits of the hash-key are between '01000' and '01111' and so on.

**Fig. 3.25. Multiple batches in hybrid hash join.**



In the hybrid hash join, the build and probe phases are performed the same number of times as the number of batches, because the inner and outer tables are stored in the same number of batches. In the first round of the build and probe phases, not only is every batch created, but also the first batches of both the inner and the outer tables are processed. On the other hand, the processing of the second and subsequent rounds needs writing and reloading to/from the temporary files, so these are costly processes. Therefore, PostgreSQL also prepares a special batch called **skew** to process many tuples more efficiently in the first round.

The skew batch stores the inner table tuples that will be joined with the outer table tuples whose MCV values of the attribute involved in the join condition are relatively large. However, this explanation may not be easy to understand, so it will be explained using a specific example.

Assume that there are two tables: `customers` and `purchase_history`. The `customers` table has two attributes: 'name' and 'address'; the `purchase_history` table has two attributes: 'customer\_name' and 'purchased\_item'. The `customers` table has 10,000 rows, and the `purchase_history` table has 1,000,000 rows. The top 10% customers have purchased 70% of all items.

Under these assumptions, let us consider how the hybrid hash join with skew performs in the first round when the query shown below is executed.

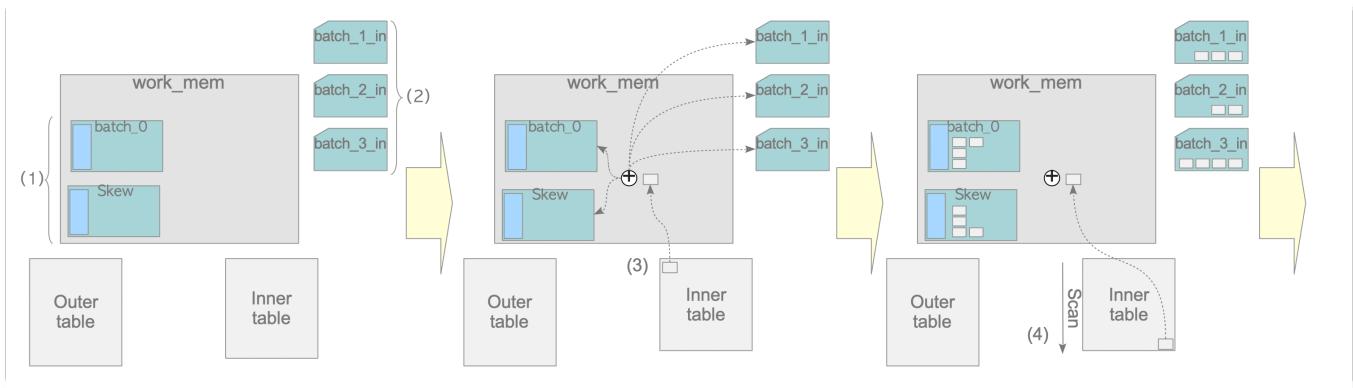
```
testdb=# SELECT * FROM customers AS c, purchase_history AS h WHERE c.name = h.customer_name;
```

If the `customers` table is inner and the `purchase_history` is outer, the top 10% of customers are stored in the skew batch using the MCV values of the `purchase_history` table. Note that the outer table's MCV values are referenced to insert the inner table tuples into the skew batch. In the probe

phase of the first round, 70% of the tuples of the outer table (`purchase_history`) will be joined with the tuples stored in the skew batch. This way, the more non-uniform the distribution of the outer table, the more tuples of the outer table can be processed in the first round.

In the following, the working of the hybrid hash join with skew is shown. Refer to Figs. 3.26 to 3.29.

**Fig. 3.26. The build phase of the hybrid hash join in the first round.**



(1) Create a batch and a skew batch on `work_mem`.

(2) Create temporary batch files for storing the inner table tuples.

In this example, three batch files are created because the inner table will be divided into four batches.

(3) Perform the build operation for the first tuple of the inner table.

The detail are described below:

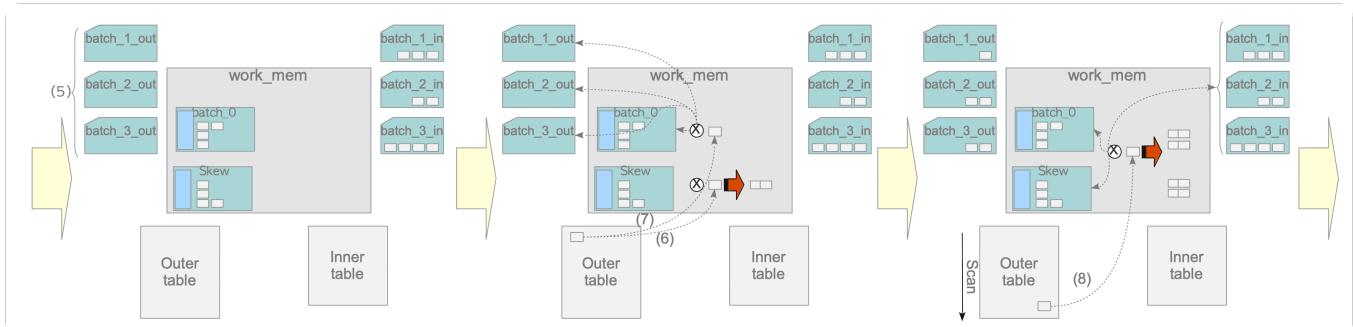
1. If the first tuple should be inserted into the skew batch, do so. Otherwise, proceed to 2.

In the example explained above, if the first tuple is one of the top 10% customers, it is inserted into the skew batch.

2. Calculate the hash key of the first tuple and then insert it into the corresponding batch.

(4) Perform the build operation for the remaining tuples of the inner table.

**Fig. 3.27. The probe phase of the hybrid hash join in the first round.**



(5) Create temporary batch files for storing the outer table tuples.

(6) If the MCV value of the first tuple is large, perform a probe operation with the skew batch.

Otherwise, proceed to (7).

In the example explained above, if the first tuple is the purchase data of the top 10% customers, it is compared with the tuples in the skew batch.

(7) Perform the probe operation of the first tuple.

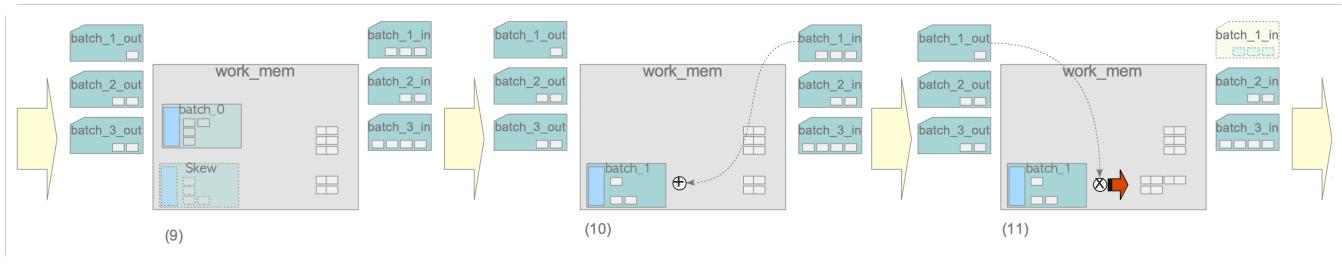
Depending on the hash-key value of the first tuple, the following process is performed:

If the first tuple belongs to Batch\_0, perform the probe operation.

Otherwise, insert into the corresponding batch.

(8) Perform the probe operation from the remaining tuples of the outer table. Note that, in this example, 70% of the tuples of the outer table have been processed by the skew in the first round without writing and reading to/from temporary files.

**Fig. 3.28. The build and probe phases in the second round.**

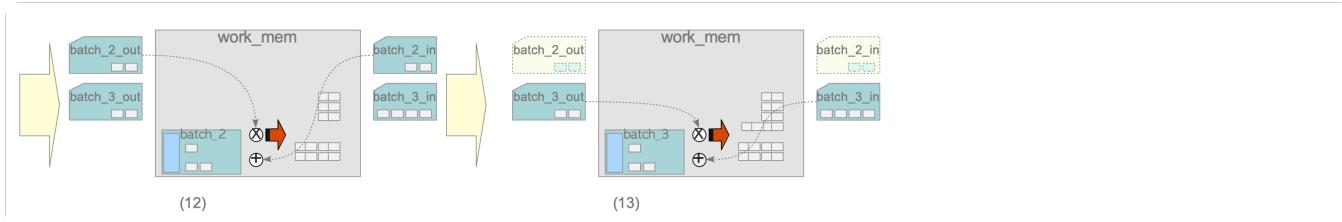


(9) Remove the skew batch and clear Batch\_0 to prepare the second round.

(10) Perform the build operation from the batch file 'batch\_1\_in'.

(11) Perform the probe operation for tuples which are stored in the batch file 'batch\_1\_out'.

**Fig. 3.29. The build and probe phases in the third and the last rounds.**



(12) Perform build and probe operations using batch files 'batch\_2\_in' and 'batch\_2\_out'.

(13) Perform build and probe operations using batch files 'batch\_3\_in' and 'batch\_3\_out'.

### 3.5.3.3. Index Scans in Hash Join

Hash join in PostgreSQL uses index scans if possible. A specific example is shown below.

```

1. testdb=# EXPLAIN SELECT * FROM pgbench_accounts AS a, pgbench_branches AS b
2. testdb-#                                     WHERE a.bid = b.bid AND a.a
3. id BETWEEN 100 AND 1000;
4. 
4. -----
5. Hash Join  (cost=1.88..51.93 rows=865 width=461)
6.   Hash Cond: (a.bid = b.bid)
7.   -> Index Scan using pgbench_accounts_pkey on pgbench_accounts a  (cost=0.4
3..47.73 rows=865 width=97)
8.         Index Cond: ((aid >= 100) AND (aid <= 1000))
9.   -> Hash  (cost=1.20..1.20 rows=20 width=364)
10.      -> Seq Scan on pgbench_branches b  (cost=0.00..1.20 rows=20 width=364)
11. (6 rows)

```

**Line 7:** In the probe phase, PostgreSQL uses the index scan when scanning the pgbench\_accounts table because there is a condition of the column 'aid' which has an index in the WHERE clause.

### **3.5.4. Join Access Paths and Join Nodes**

### **3.5.4.1. Join Access Paths**

The `JoinPath` structure is an access path for the nested loop join. Other join access paths, such as `MergePath` and `HashPath`, are based on it.

All join access paths are illustrated below, without explanation.

```

/*
 * We might need additional join types someday.
 */
} JoinType;

/*
 * All join-type paths share these fields.
 */

typedef struct JoinPath
{
    pg_node_attr(abstract)

    Path          path;
    JoinType      jointype;
    bool          inner_unique; /* each outer tuple provably matches no more
                                 * than one inner tuple */
    Path          *outerjoinpath; /* path for the outer side of the join */
    Path          *innerjoinpath; /* path for the inner side of the join */
    List          *joinrestrictinfo; /* RestrictInfos to apply to join */

    /*
     * See the notes for RelOptInfo and ParamPathInfo to understand why
     * joinrestrictinfo is needed in JoinPath, and can't be merged into the
     * parent RelOptInfo.
     */
} JoinPath;
/* 
 * A mergejoin path has these fields.
 *
 * Unlike other path types, a MergePath node doesn't represent just a single
 * run-time plan node: it can represent up to four. Aside from the MergeJoin
 * node itself, there can be a Sort node for the outer input, a Sort node
 * for the inner input, and/or a Material node for the inner input. We could
 * represent these nodes by separate path nodes, but considering how many
 * different merge paths are investigated during a complex join problem,
 * it seems better to avoid unnecessary malloc overhead.
 *
 * path_mergeclauses lists the clauses (in the form of RestrictInfos)
 * that will be used in the merge.
 *
 * Note that the mergeclauses are a subset of the parent relation's
 * restriction-clause list. Any join clauses that are not mergejoinable
 * appear only in the parent's restrict list, and must be checked by a
 * qpqual at execution time.
 *
 *outersortkeys (resp. innersortkeys) is NIL if the outer path
 * (resp. inner path) is already ordered appropriately for the
 * mergejoin. If it is not NIL then it is a PathKeys list describing
 * the ordering that must be created by an explicit Sort node.
 *
 * skip_mark_restore is true if the executor need not do mark/restore calls.
 * Mark/restore overhead is usually required, but can be skipped if we know
 * that the executor need find only one match per outer tuple, and that the
 * mergeclauses are sufficient to identify a match. In such cases the

```

```

* executor can immediately advance the outer relation after processing a
* match, and therefore it need never back up the inner relation.
*
* materialize_inner is true if a Material node should be placed atop the
* inner input. This may appear with or without an inner Sort step.
*/

```

```

typedef struct MergePath
{
    JoinPath      jpath;
    List         *path_mergeclauses; /* join clauses to be used for merge */
    List         *outersortkeys;     /* keys for explicit sort, if any */
    List         *innersortkeys;     /* keys for explicit sort, if any */
    bool          skip_mark_restore; /* can executor skip mark/restore? */
    bool          materialize_inner; /* add Materialize to inner? */
} MergePath;

```

```

/*
* A hashjoin path has these fields.
*
* The remarks above for mergeclauses apply for hashclauses as well.
*
* Hashjoin does not care what order its inputs appear in, so we have
* no need for sortkeys.
*/

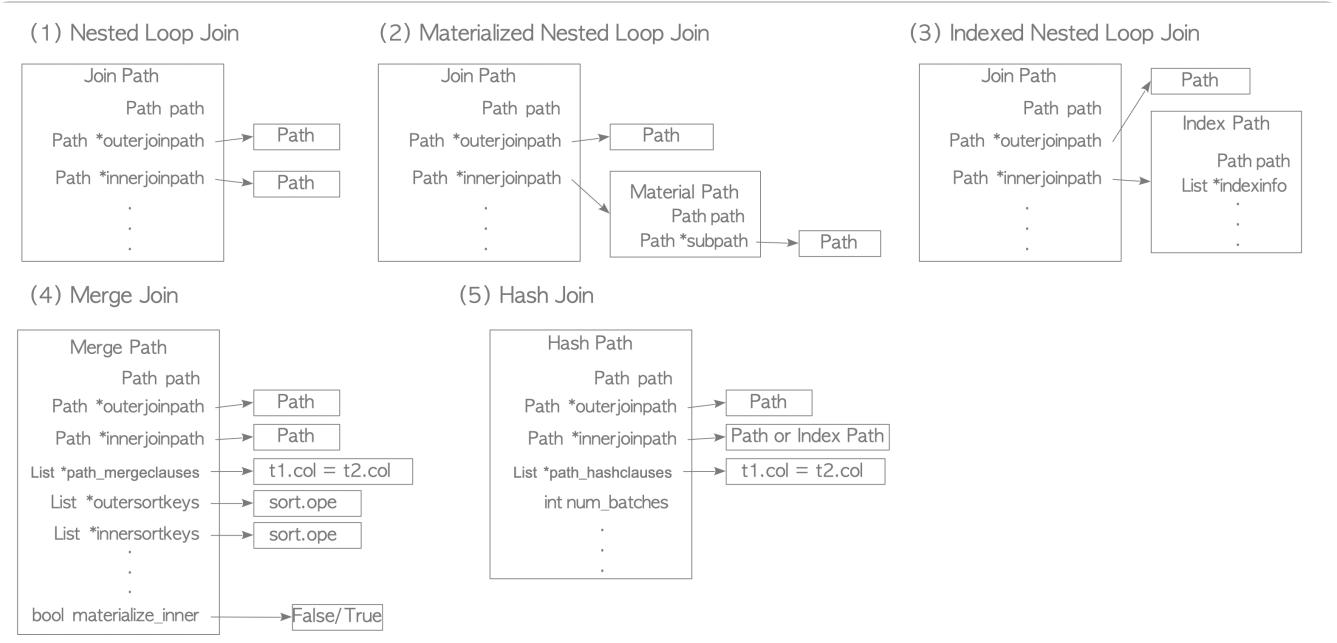
```

```

typedef struct HashPath
{
    JoinPath      jpath;
    List         *path_hashclauses; /* join clauses used for hashing */
    int           num_batches;     /* number of batches expected */
    Cardinality   inner_rows_total; /* total inner rows expected */
} HashPath;

```

**Fig. 3.30. Join access paths.**



### 3.5.4.2. Join Nodes

This subsection shows the three join nodes, `NestedLoopNode`, `MergeJoinNode` and `HashJoinNode`, without explanation. They are all based on the `JoinNode`.

```

/*
 *          Join node
 *
 * jointype:    rule for joining tuples from left and right subtrees
 * inner_unique each outer tuple can match to no more than one inner tuple
 * joinqual:   qual conditions that came from JOIN/ON or JOIN/USING
 *                         (plan.qual contains conditions that came from WHERE)
 *
 * When jointype is INNER, joinqual and plan.qual are semantically
 * interchangeable. For OUTER jointypes, the two are *not* interchangeable;
 * only joinqual is used to determine whether a match has been found for
 * the purpose of deciding whether to generate null-extended tuples.
 * (But plan.qual is still applied before actually returning a tuple.)
 * For an outer join, only joinquals are allowed to be used as the merge
 * or hash condition of a merge or hash join.
 *
 * inner_unique is set if the joinquals are such that no more than one inner
 * tuple could match any given outer tuple. This allows the executor to
 * skip searching for additional matches. (This must be provable from just
 * the joinquals, ignoring plan.qual, due to where the executor tests it.)
 *
 */
typedef struct Join
{
    pg_node_attr(abstract)

    Plan          plan;
    JoinType     jointype;
    bool         inner_unique;
    List         *joinqual;           /* JOIN quals (in addition to plan.qual) */
} Join;

/*
 *          nest loop join node
 *
 * The nestParams list identifies any executor Params that must be passed
 * into execution of the inner subplan carrying values from the current row
 * of the outer subplan. Currently we restrict these values to be simple
 * Vars, but perhaps someday that'd be worth relaxing. (Note: during plan
 * creation, the paramval can actually be a PlaceHolderVar expression; but it
 * must be a Var with varno OUTER_VAR by the time it gets to the executor.)
 *
 */
typedef struct NestLoop
{
    Join          join;
    List         *nestParams;        /* list of NestLoopParam nodes */
} NestLoop;

typedef struct NestLoopParam
{
    pg_node_attr(no_equal, no_query_jumble)

    NodeTag       type;
    int           paramno;          /* number of the PARAM_EXEC P
aram to set */
    Var           *paramval;        /* outer-relation Var to assign to Pa
ram */
} NestLoopParam;

/*
 *          merge join node

```

```

/*
 * The expected ordering of each mergeable column is described by a btree
 * opfamily OID, a collation OID, a direction (BTLessStrategyNumber or
 * BTGreaterStrategyNumber) and a nulls-first flag. Note that the two sides
 * of each mergeclause may be of different datatypes, but they are ordered the
 * same way according to the common opfamily and collation. The operator in
 * each mergeclause must be an equality operator of the indicated opfamily.
 */
typedef struct MergeJoin
{
    Join          join;

    /* Can we skip mark/restore calls? */
    bool         skip_mark_restore;

    /* mergeclauses as expression trees */
    List        *mergeclauses;

    /* these are arrays, but have the same length as the mergeclauses list: */
    /* per-clause OIDs of btree opfamilies */
    Oid          *mergeFamilies pg_node_attr(array_size(mergeclauses));

    /* per-clause OIDs of collations */
    Oid          *mergeCollations pg_node_attr(array_size(mergeclauses));

    /* per-clause ordering (ASC or DESC) */
    int          *mergeStrategies pg_node_attr(array_size(mergeclauses));

    /* per-clause nulls ordering */
    bool         *mergeNullsFirst pg_node_attr(array_size(mergeclauses));
} MergeJoin;
/* -----
 *          hash join node
 * -----
 */
typedef struct HashJoin
{
    Join          join;
    List        *hashclauses;
    List        *hashoperators;
    List        *hashcollations;

    /*
     * List of expressions to be hashed for tuples from the outer plan, to
     * perform lookups in the hashtable over the inner plan.
     */
    List        *hashkeys;
} HashJoin;

```

## 3.6. Creating the Plan Tree of Multiple-Table Query

In this section, the process of creating a plan tree of a multiple-table query is explained.

### 3.6.1. Preprocessing

The `subquery_planner()` function defined in `planner.c` invokes preprocessing. The preprocessing for single-table queries has already been described in Section 3.3.1. In this subsection, the preprocessing for a multiple-table query will be described. However, only some parts will be described, as there are many.

## 1. Planning and Converting CTE

If there are `WITH` lists, the planner processes each `WITH` query by the `SS_process_ctes()` function.

## 2. Pulling Subqueries Up

If the `FROM` clause has a subquery and it does not have `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT` or `DISTINCT` clauses, and also it does not use `INTERSECT` or `EXCEPT`, the planner converts to a join form by the `pull_up_subqueries()` function. For example, the query shown below which contains a subquery in the `FROM` clause can be converted to a natural join query. Needless to say, this conversion is done in the query tree.

```
testdb=# SELECT * FROM tbl_a AS a, (SELECT * FROM tbl_b) as b WHERE a.id = b.id;  
                  ↓  
testdb=# SELECT * FROM tbl_a AS a, tbl_b as b WHERE a.id = b.id;
```

## 3. Transforming an Outer Join to an Inner Join

The planner transforms an outer join query to an inner join query if possible.

## 3.6.2. Getting the Cheapest Path

To get the optimal plan tree, the planner has to consider all the combinations of indexes and join methods. This is a very expensive process and it will be infeasible if the number of tables exceeds a certain level due to a combinatorial explosion. Fortunately, if the number of tables is smaller than around 12, the planner can get the optimal plan by applying dynamic programming. Otherwise, the planner uses the *genetic algorithm*. Refer to the  below.

### Genetic Query Optimizer

When a query joining many tables is executed, a huge amount of time will be needed to optimize the query plan. To deal with this situation, PostgreSQL implements an interesting feature: the **Genetic Query Optimizer**. This is a kind of approximate algorithm to determine a reasonable plan within a reasonable time. Hence, in the query optimization stage, if the number of the joining tables is higher than the threshold specified by the parameter `geqo_threshold` (the default is 12), PostgreSQL generates a query plan using the genetic algorithm.

Determination of the optimal plan tree by dynamic programming can be explained by the following steps:

*Level = 1*

Get the cheapest path for each table. The cheapest path is stored in the respective `RelOptInfo`.

*Level = 2*

Get the cheapest path for each combination of two tables.

For example, if there are two tables, A and B, get the cheapest join path of tables A and B. This is the final answer.

In the following, the `RelOptInfo` of two tables is represented by {A, B}.

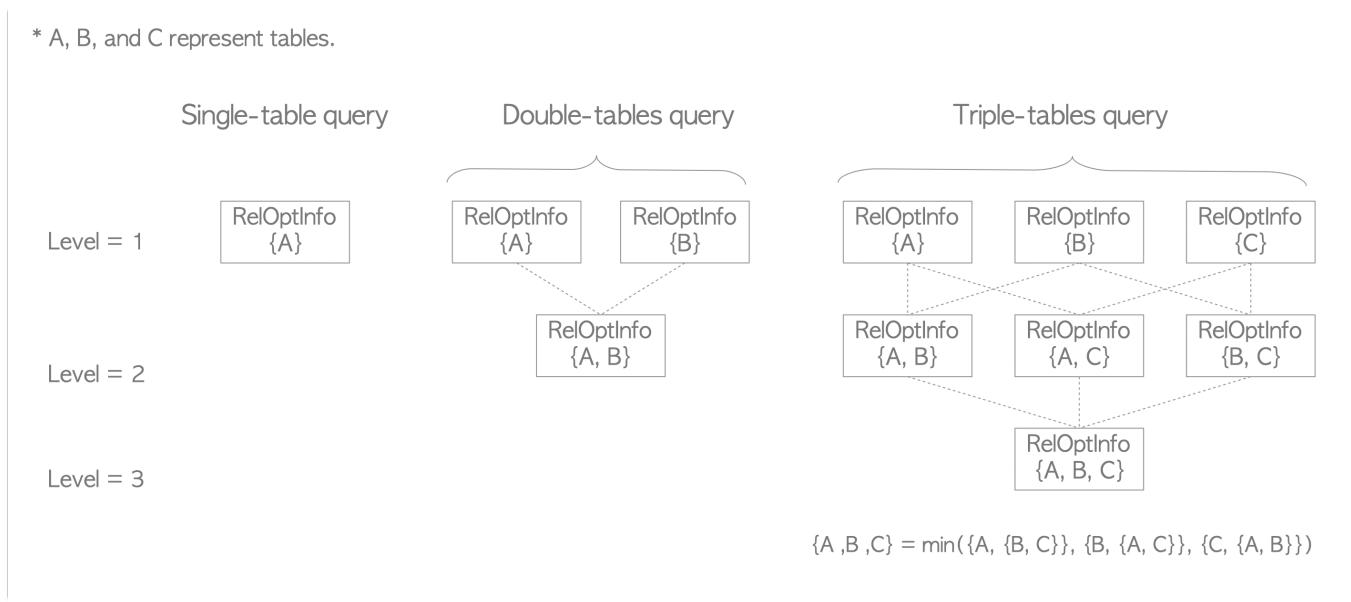
If there are three tables, get the cheapest path for each of {A, B}, {A, C}, and {B, C}.

*Level = 3 and higher*

Continue the same process until the level that equals the number of tables is reached.

This way, the cheapest paths of the partial problems are obtained at each level and are used to get the upper level's calculation. This makes it possible to calculate the cheapest plan tree efficiently.

**Fig. 3.31. How to get the cheapest access path using dynamic programming.**



In the following, the process of how the planner gets the cheapest plan of the following query is described.

```
testdb=# \d tbl_a
Table "public.tbl_a"
Column | Type | Modifiers
-----+-----+
id    | integer | not null
data  | integer |
Indexes:
"tbl_a_pkey" PRIMARY KEY, btree (id)

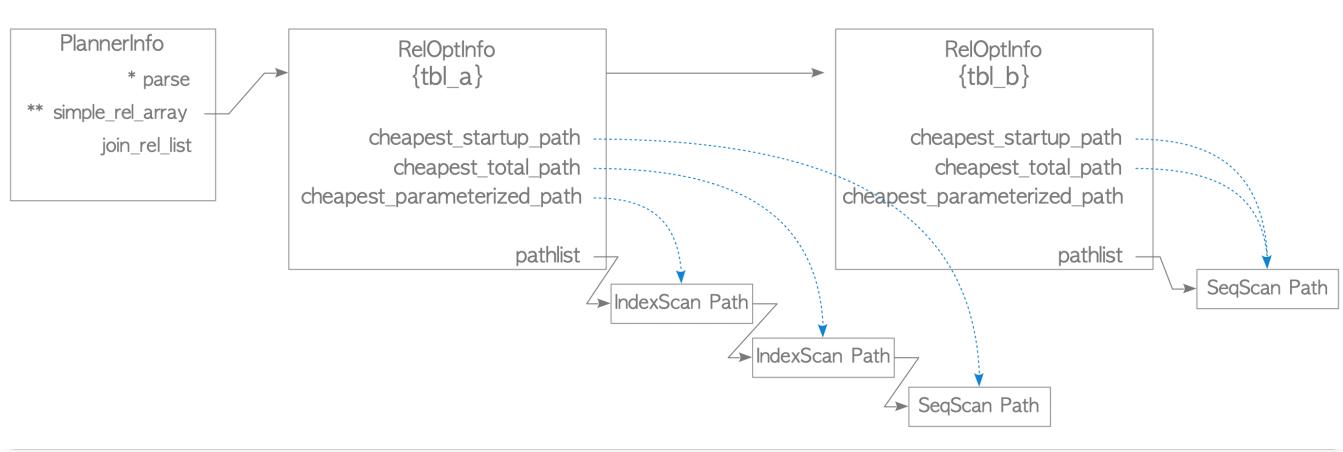
testdb=# \d tbl_b
Table "public.tbl_b"
Column | Type | Modifiers
-----+-----+
id    | integer |
data  | integer |

testdb=# SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND b.data < 400;
```

### 3.6.2.1. Processing in Level 1

In Level 1, the planner creates a RelOptInfo structure and estimates the cheapest costs for each relation in the query. The RelOptInfo structures are added to the simple\_rel\_array of the PlannerInfo of this query.

**Fig. 3.32. The PlannerInfo and RelOptInfo after processing in Level 1.**



The RelOptInfo of `tbl_a` has three access paths, which are added to the `pathlist` of the RelOptInfo. Each access path is linked to a cheapest cost path: the *cheapest start-up (cost) path*, the *cheapest total (cost) path*, and the *cheapest parameterized (cost) path*. The cheapest start-up and total cost paths are obvious, so the cost of the cheapest parameterized index scan path will be described.

As described in Section 3.5.1.3, the planner considers the use of the parameterized path for the indexed nested loop join (and rarely the indexed merge join with an outer index scan). The cheapest parameterized cost is the cheapest cost of the estimated parameterized paths.

The RelOptInfo of `tbl_b` only has a sequential scan access path because `tbl_b` does not have a related index.

### 3.6.2.2. Processing in Level 2

In level 2, a RelOptInfo structure is created and added to the `join_rel_list` of the PlannerInfo. Then, the costs of all possible join paths are estimated, and the best access path, whose total cost is the cheapest, is selected. The RelOptInfo stores the best access path as the *cheapest total cost path*. See Fig. 3.33.

**Fig. 3.33. The PlannerInfo and RelOptInfo after processing in Level 2.**

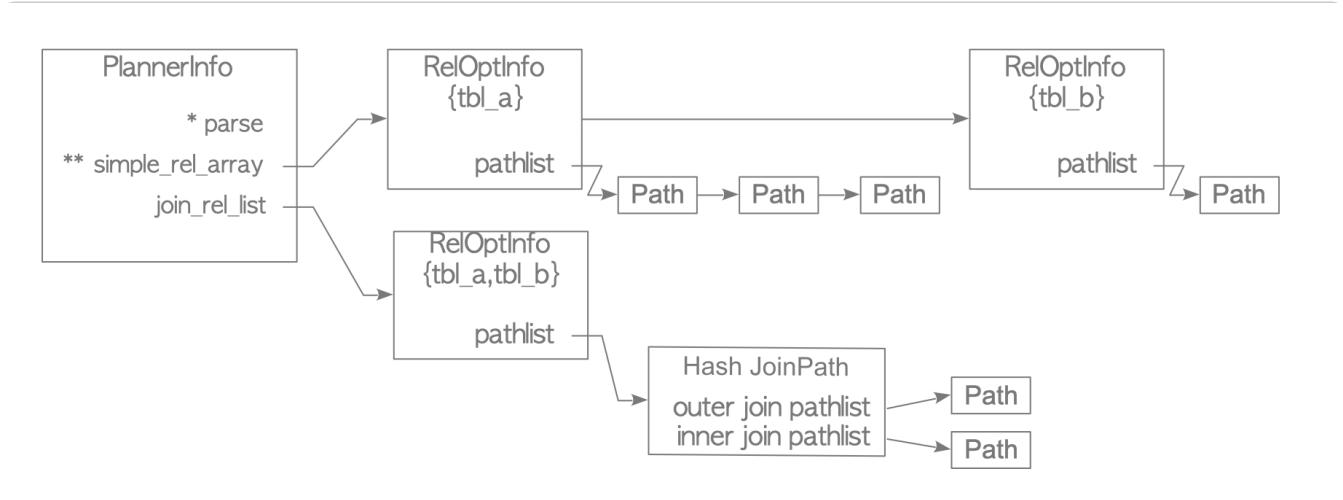


Table 3.1 shows all combinations of join access paths in this example. The query of this example is an equi-join type, so all three join methods are estimated. For convenience, some notations of access paths are introduced:

- *SeqScanPath(table)* means the sequential scan path of table.
- *Materialized->SeqScanPath(table)* means the materialized sequential scan path of a table.

- *IndexScanPath(table, attribute)* means the index scan path by the attribute of the a table.
- *ParameterizedIndexScanPath(table, attribute1, attribute2)* means the parameterized index path by the attribute1 of the table, and it is parameterized by attribute2 of the outer table.

**Table 3.1: All combinations of join access paths in this example**

Outer Path	Inner Path	
Nested Loop Join		
1 SeqScanPath(tbl_a)	SeqScanPath(tbl_b)	
2 SeqScanPath(tbl_a)	Materialized->SeqScanPath(tbl_b)	Materialized nested loop join
3 IndexScanPath(tbl_a,id)	SeqScanPath(tbl_b)	Nested loop join with outer index scan
4 IndexScanPath(tbl_a,id)	Materialized->SeqScanPath(tbl_b)	Materialized nested loop join with outer index scan
5 SeqScanPath(tbl_b)	SeqScanPath(tbl_a)	
6 SeqScanPath(tbl_b)	Materialized->SeqScanPath(tbl_a)	Materialized nested loop join
7 SeqScanPath(tbl_b)	ParametalizedIndexScanPath(tbl_a, id, tbl_b.id)	Indexed nested loop join
Merge Join		
1 SeqScanPath(tbl_a)	SeqScanPath(tbl_b)	
2 IndexScanPath(tbl_a,id)	SeqScanPath(tbl_b)	Merge join with outer index scan
3 SeqScanPath(tbl_b)	SeqScanPath(tbl_a)	
Hash Join		
1 SeqScanPath(tbl_a)	SeqScanPath(tbl_b)	
2 SeqScanPath(tbl_b)	SeqScanPath(tbl_a)	

For example, in the nested loop join, seven join paths are estimated. The first one indicates that the outer and inner paths are the sequential scan paths of `tbl_a` and `tbl_b`, respectively. The second indicates that the outer path is the sequential scan path of `tbl_a` and the inner path is the materialized sequential scan path of `tbl_b`. And so on.

The planner finally selects the cheapest access path from the estimated join paths, and the cheapest path is added to the pathlist of the `RelOptInfo` `{tbl_a,tbl_b}`. See Fig. 3.33.

In this example, as shown in the result of EXPLAIN below, the planner selects the hash join whose inner and outer tables are `tbl_b` and `tbl_c`.

```

1. testdb=# EXPLAIN  SELECT * FROM tbl_b AS b, tbl_c AS c WHERE c.id = b.id AND b.da
   ta < 400;
2.                                     QUERY PLAN
3. -----
4. Hash Join  (cost=90.50..277.00 rows=400 width=16)
5.   Hash Cond: (c.id = b.id)
6.     -> Seq Scan on tbl_c c  (cost=0.00..145.00 rows=10000 width=8)

```

```

7.      -> Hash (cost=85.50..85.50 rows=400 width=8)
8.          -> Seq Scan on tbl_b b (cost=0.00..85.50 rows=400 width=8)
9.              Filter: (data < 400)
10.             (6 rows)

```

### 3.6.3. Getting the Cheapest Path of a Triple-Table Query

Obtaining the cheapest path of a query involving three tables is as follows:

```

testdb=# \d tbl_a
  Table "public.tbl_a"
 Column | Type    | Modifiers
-----+-----+
 id    | integer |
 data  | integer |

testdb=# \d tbl_b
  Table "public.tbl_b"
 Column | Type    | Modifiers
-----+-----+
 id    | integer |
 data  | integer |

testdb=# \d tbl_c
  Table "public.tbl_c"
 Column | Type    | Modifiers
-----+-----+
 id    | integer | not null
 data  | integer |

Indexes:
 "tbl_c_pkey" PRIMARY KEY, btree (id)

testdb=# SELECT * FROM tbl_a AS a, tbl_b AS b, tbl_c AS c
testdb#           WHERE a.id = b.id AND b.id = c.id AND a.data < 40;

```

#### Level 1:

The planner estimates the cheapest paths of all tables and stores this information in the corresponding RelOptInfo objects: {tbl\_a}, {tbl\_b}, and {tbl\_c}.

#### Level 2:

The planner picks all the combinations of pairs of the three tables and estimates the cheapest path for each combination. The planner then stores the information in the corresponding RelOptInfo objects: {tbl\_a, tbl\_b}, {tbl\_b, tbl\_c}, and {tbl\_a, tbl\_c}.

#### Level 3:

The planner finally gets the cheapest path using the already obtained RelOptInfo objects. More precisely, the planner considers three combinations of RelOptInfo objects: {tbl\_a, {tbl\_b, tbl\_c}}, {tbl\_b, {tbl\_a, tbl\_c}}, and {tbl\_c, {tbl\_a, tbl\_b}}, because

$$\{tbl_a, tbl_b, tbl_c\} = \min(\{tbl_a, \{tbl_b, tbl_c\}\}, \{tbl_b, \{tbl_a, tbl_c\}\}, \{tbl_c, \{tbl_a, tbl_b\}\}).$$

The planner then estimates the costs of all possible join paths in them.

In the RelOptInfo object {tbl\_c, {tbl\_a, tbl\_b}}, the planner estimates all the combinations of tbl\_c and the cheapest path of {tbl\_a, tbl\_b}, which is the hash join whose inner and outer tables are

tbl\_a and tbl\_b, respectively, in this example. The estimated join paths will contain three kinds of join paths and their variations, such as those shown in the previous subsection, that is, the nested loop join and its variations, the merge join and its variations, and the hash join.

The planner processes the RelOptInfo objects {tbl\_a, {tbl\_b, tbl\_c}} and {tbl\_b, {tbl\_a, tbl\_c}} in the same way and finally selects the cheapest access path from all the estimated paths.

The result of the EXPLAIN command of this query is shown below:

```

1. testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b, tbl_c AS c
2. testdb-# WHERE a.id = b.id AND b.id = c.id AND a.data < 40;
3.                                     QUERY PLAN
4. -----
5. Nested Loop  (cost=170.77..269.94 rows=20 width=24)
6.   Join Filter: (a.id = c.id)
7.   -> Hash Join  (cost=170.49..262.44 rows=20 width=16) } Outer relation of Indexed Nested Loop Join
8.     Hash Cond: (b.id = a.id)
9.     -> Seq Scan on tbl_b b  (cost=0.00..73.00 rows=5000 width=8)
10.    -> Hash  (cost=170.00..170.00 rows=39 width=8)
11.      -> Seq Scan on tbl_a a  (cost=0.00..170.00 rows=39 width=8)
12.          Filter: (data < 40)
13.      -> Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..0.36 rows=1 width=8)
14.          Index Cond: (id = b.id)
15. (10 rows)

```

The outermost join is the indexed nested loop join (Line 5). The inner parameterized index scan is shown in line 13, and the outer relation is the result of the hash join whose inner and outer tables are tbl\_b and tbl\_a, respectively (lines 7-12). Therefore, the executor first executes the hash join of tbl\_a and tbl\_b and then executes the indexed nested loop join.

## References

- [1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, "Database System Concepts", McGraw-Hill Education, ISBN-13: 978-0073523323
- [2] Thomas M. Connolly, and Carolyn E. Begg, "Database Systems", Pearson, ISBN-13: 978-0321523068

[◀ Back to Part 1, Part 2](#)

# Chapter 4

## Foreign Data Wrappers and Parallel Query

This chapter will describe two technically interesting and practical features: Foreign Data Wrappers(FDW) and Parallel Query.

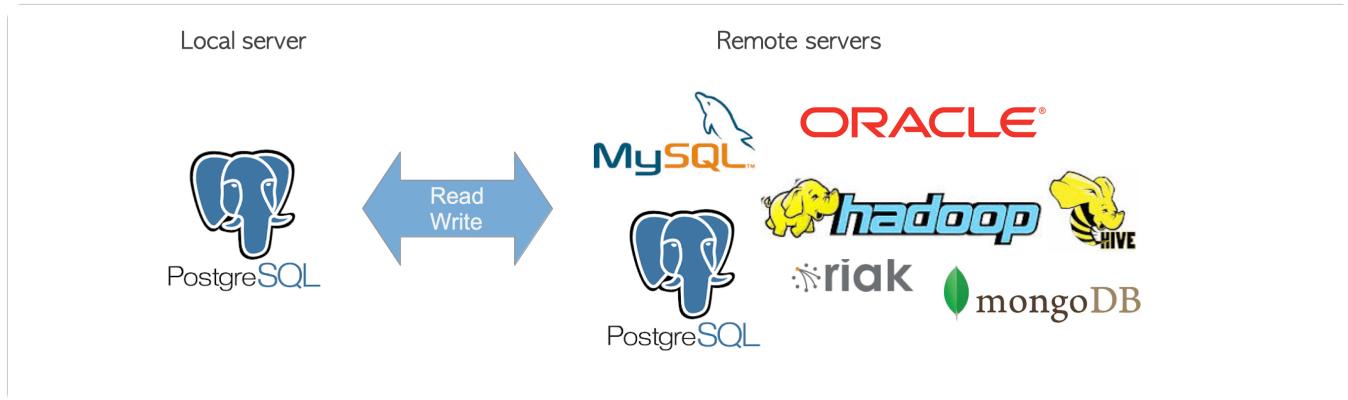
Currently, Section 4.1. FDW is only provided; Section 4.2. Parallel Query is under construction.

### 4.1. Foreign Data Wrappers (FDW)

In 2003, the SQL standard added a specification to access remote data called SQL Management of External Data (SQL/MED). PostgreSQL has been developing this feature since version 9.1 to realize a portion of SQL/MED.

In SQL/MED, a table on a remote server is called a *foreign table*. PostgreSQL's **Foreign Data Wrappers (FDW)** use SQL/MED to manage foreign tables, which are similar to local tables.

**Fig. 4.1. Basic concept of FDW.**



After installing the necessary extension and making the appropriate settings, you can access foreign tables on remote servers. For example, suppose there are two remote servers, namely PostgreSQL and MySQL, which have the `foreign_pg_tbl` and `foreign_my_tbl` tables, respectively. In this example, you can access the foreign tables from the local server by issuing the following SELECT queries:

```
localdb=# -- foreign_pg_tbl is on the remote postgresql server.  
localdb# SELECT count(*) FROM foreign_pg_tbl;  
count  
-----  
20000
```

```
localdb=# -- foreign_my_tbl is on the remote mysql server.  
localdb# SELECT count(*) FROM foreign_my_tbl;  
count
```

```
-----  
10000
```

You can also execute join operations with foreign tables stored in different servers, just like you would with local tables.

```
localdb=# SELECT count(*) FROM foreign_pg_tbl AS p, foreign_my_tbl AS m WHERE p.id = m.id;  
count  
-----  
10000
```

Many FDW extensions have been developed and listed on the Postgres wiki. However, almost all of them are not properly maintained, with the exception of `postgres_fdw`, which is officially developed and maintained by the PostgreSQL Global Development Group as an extension to access a remote PostgreSQL server.

PostgreSQL's FDW is described in detail in the following two sections. Section 4.1.1 provides an overview of FDW in PostgreSQL. Section 4.1.2 describes how the `postgres_fdw` extension performs.

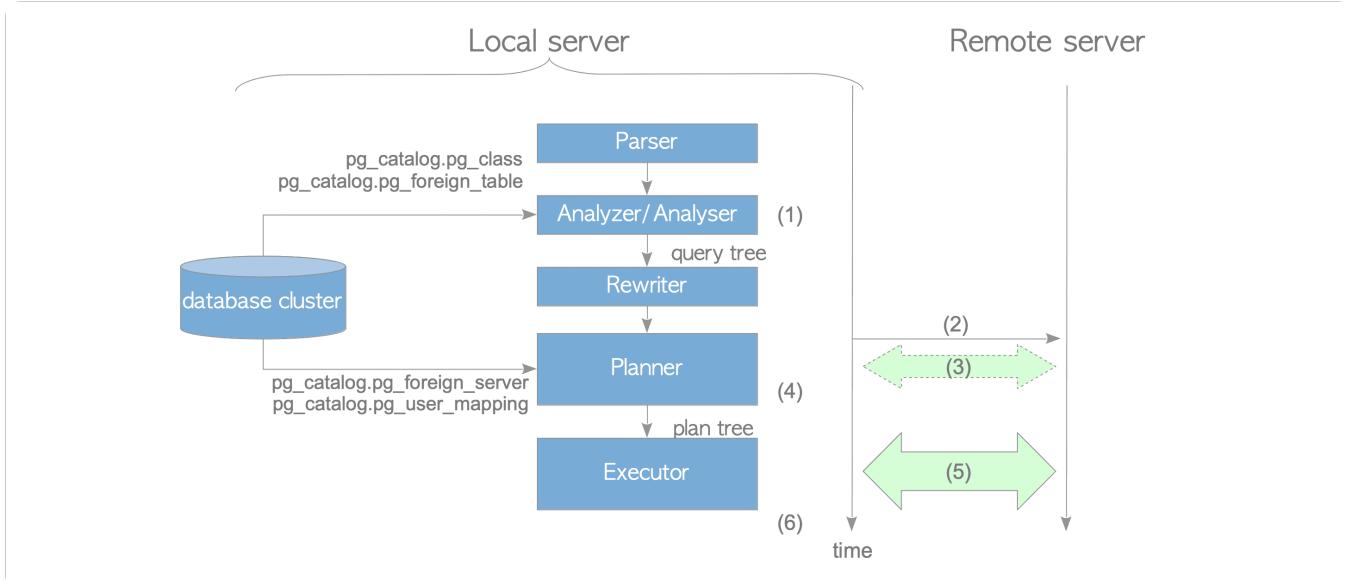
## 4.1.1. Overview

To use the FDW feature, you need to install the appropriate extension and execute setup commands such as `CREATE FOREIGN TABLE`, `CREATE SERVER` and `CREATE USER MAPPING` (for details, refer to the [official document](#)).

After providing the appropriate setting, the functions defined in the extension are invoked during query processing to access the foreign tables.

Fig.4.2 briefly describes how FDW performs in PostgreSQL.

**Fig. 4.2. How FDWs perform.**



(1) The analyzer/analyser creates the query tree of the input SQL.

(2) The planner (or executor) connects to the remote server.

(3) If the `use_remote_estimate` option is `on` (the default is `off`), the planner executes EXPLAIN commands to estimate the cost of each plan path.

- (4) The planner creates a plain text SQL statement from the plan tree which is internally called *deparsing*.
- (5) The executor sends the plain text SQL statement to the remote server and receives the result.

The executor then processes the received data if necessary. For example, if a multi-table query is executed, the executor performs the join processing of the received data and other tables.

The details of each processing are described in the following sections.

#### **4.1.1.1. Creating a Query Tree**

The analyzer/analyser creates the query tree of the input SQL using the definitions of the foreign tables, which are stored in the `pg_catalog.pg_class` and `pg_catalog.pg_foreign_table` catalogs using the `CREATE FOREIGN TABLE` or `IMPORT FOREIGN SCHEMA` command.

#### **4.1.1.2. Connecting to the Remote Server**

To connect to the remote server, the planner (or executor) uses the appropriate library to connect to the remote database server. For example, to connect to the remote PostgreSQL server, the `postgres_fdw` uses the `libpq` library. To connect to the MySQL server, the `mysql_fdw` extension, which is developed by EnterpriseDB, uses the `libmysqlclient` library.

The connection parameters, such as username, server's IP address and port number, are stored in the `pg_catalog.pg_user_mapping` and `pg_catalog.pg_foreign_server` catalogs using the `CREATE USER MAPPING` and `CREATE SERVER` commands.

#### **4.1.1.3. Creating a Plan Tree Using EXPLAIN Commands (Optional)**

PostgreSQL's FDW supports the ability to obtain statistics of foreign tables to estimate the plan tree of a query. Some FDW extensions, such as `postgres_fdw`, `mysql_fdw`, `tds_fdw`, and `jdbc2_fdw`, use these statistics.

If the `use_remote_estimate` option is set to `on` using the `ALTER SERVER` command, the planner queries the cost of plans to the remote server by executing the `EXPLAIN` command. Otherwise, the embedded constant values are used by default.

```
localdb=# ALTER SERVER remote_server_name OPTIONS (use_remote_estimate 'on');
```

Although some extensions use the values of the `EXPLAIN` command, only `postgres_fdw` can reflect the results of the `EXPLAIN` commands because PostgreSQL's `EXPLAIN` command returns both the start-up and total costs.

The results of the `EXPLAIN` command cannot be used by other DBMS FDW extensions for planning. For example, MySQL's `EXPLAIN` command only returns the estimated number of rows. However, PostgreSQL's planner needs more information to estimate the cost as described in Chapter 3.

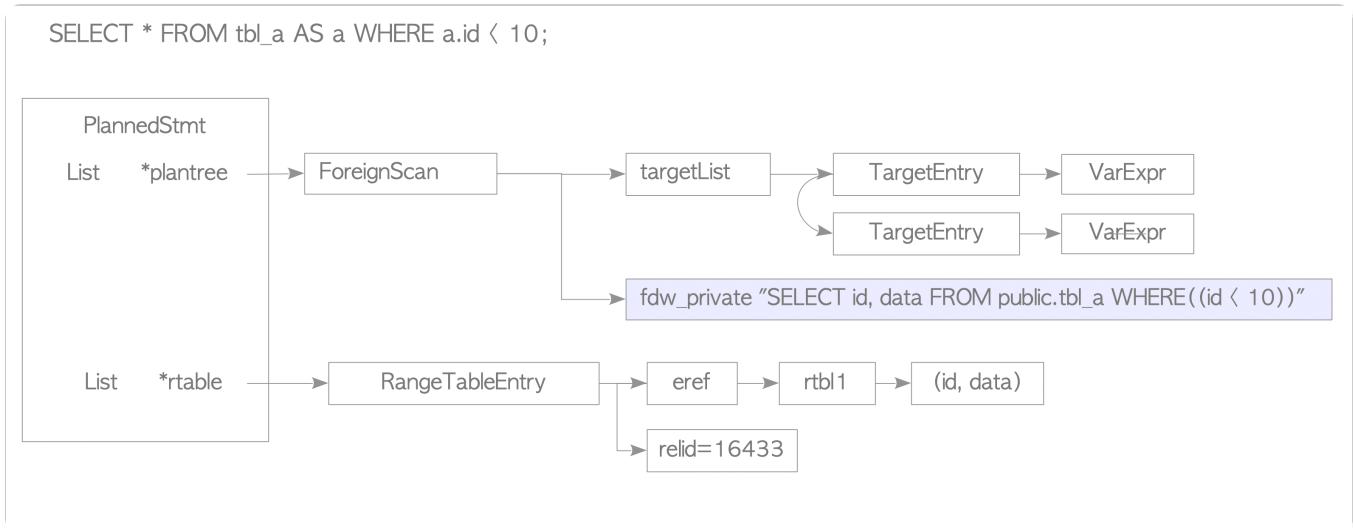
#### **4.1.1.4. Deparsing**

To generate the plan tree, the planner creates a plain text SQL statement from the plan tree's scan paths of the foreign tables. For example, Fig. 4.3 shows the plan tree of the following `SELECT` statement:

```
localdb=# SELECT * FROM tbl_a AS a WHERE a.id < 10;
```

Fig.4.3 shows that the ForeignScan node, which is linked from the plan tree of the PlannedStmt, stores a plain SELECT text. Here, postgres\_fdw recreates a plain SELECT text from the query tree that has been created by parsing and analysing, which is called **deparsing** in PostgreSQL.

**Fig. 4.3. Example of the plan tree that scans a foreign table.**



The use of mysql\_fdw recreates a SELECT text for MySQL from the query tree. The use of redis\_fdw or rw\_redis\_fdw creates a SELECT command.

#### 4.1.1.5. Sending SQL Statements and Receiving Result

After deparsing, the executor sends the deparsed SQL statements to the remote server and receives the result.

The method for sending the SQL statements to the remote server depends on the developer of each extension. For example, mysql\_fdw sends the SQL statements without using a transaction. The typical sequence of SQL statements to execute a SELECT query in mysql\_fdw is shown below (Fig. 4.4).

- (5-1) Set the `SQL_MODE` to '`ANSI_QUOTES`'.
- (5-2) Send a `SELECT` statement to the remote server.
- (5-3) Receive the result from the remote server.

Here, mysql\_fdw converts the result to readable data by PostgreSQL.

All FDW extensions implement the feature that converts the result to PostgreSQL readable data.

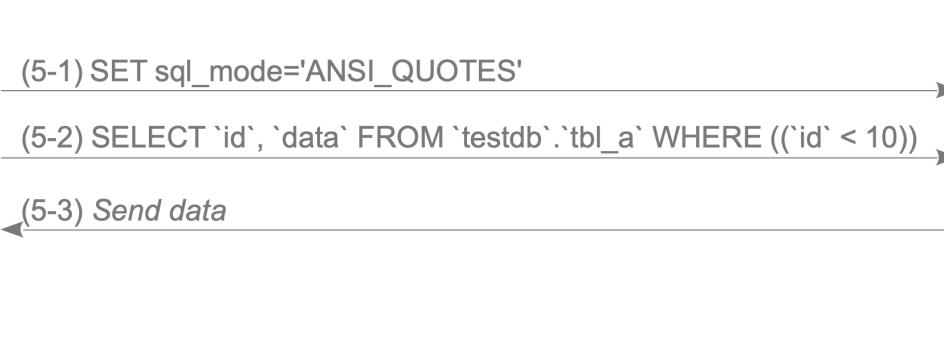
**Fig. 4.4. Typical sequence of SQL statements to execute a SELECT query in mysql\_fdw**



Local server



Remote server



The actual log of the remote server can be found [here](#); the statements received by the remote server are shown.

In `postgres_fdw`, the sequence of SQL commands is more complicated. The typical sequence of SQL statements to execute a SELECT query in `postgres_fdw` is shown below (Fig. 4.5).

(5-1) Start the remote transaction.

The default remote transaction isolation level is REPEATABLE READ; if the isolation level of the local transaction is set to SERIALIZABLE, the remote transaction is also set to SERIALIZABLE.

(5-2)-(5-4) Declare a cursor.

The SQL statement is basically executed as a cursor.

(5-5) Execute FETCH commands to obtain the result.

By default, 100 rows are fetched by the FETCH command.

(5-6) Receive the result from the remote server.

(5-7) Close the cursor.

(5-8) Commit the remote transaction.

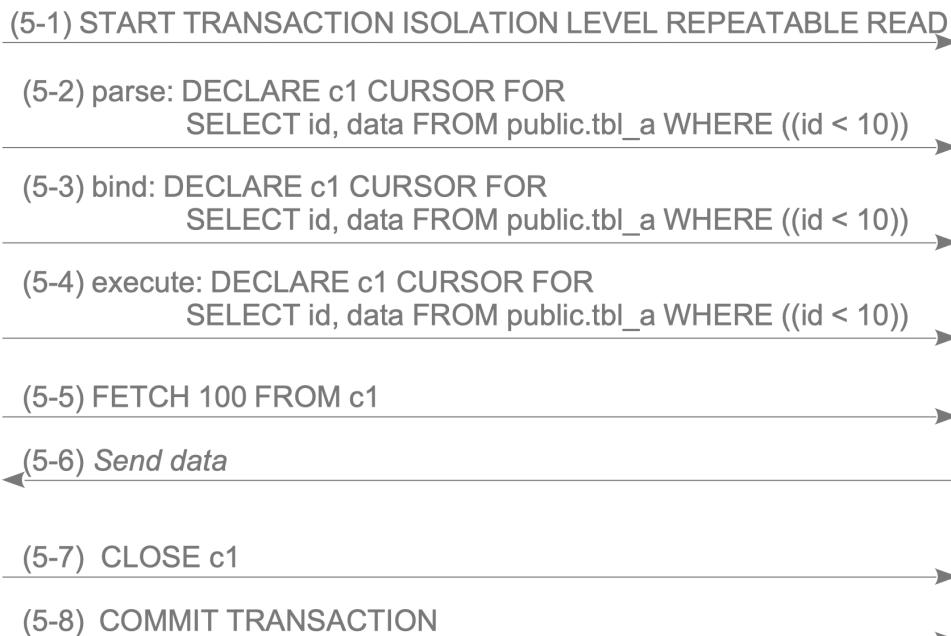
**Fig. 4.5. Typical sequence of SQL statements to execute a SELECT query in `postgres_fdw`.**



Local server



Remote server



The actual log of the remote server can be found [here](#).

### **ⓘ The default remote transaction isolation level in postgres\_fdw.**

The explanation for why the default remote transaction isolation level is REPEATABLE READ is provided in the official document.

The remote transaction uses the SERIALIZABLE isolation level when the local transaction has the SERIALIZABLE isolation level; otherwise it uses the REPEATABLE READ isolation level. This choice ensures that if a query performs multiple table scans on the remote server, it will get snapshot-consistent results for all the scans. A consequence is that successive queries within a single transaction will see the same data from the remote server, even if concurrent updates are occurring on the remote server due to other activities.

## **4.1.2. How the Postgres\_fdw Extension Performs**

The postgres\_fdw extension is a special module that is officially maintained by the PostgreSQL Global Development Group. Its source code is included in the PostgreSQL source code tree.

postgres\_fdw is gradually improved. Table 4.1 presents the release notes related to postgres\_fdw from the official document.

**Table 4.1 Release notes related to postgres\_fdw (cited from the official document).**

Version	Description
9.3	postgres_fdw module is released.
9.6	Consider performing sorts on the remote server. Consider performing joins on the remote server. When feasible, perform UPDATE or DELETE entirely on the remote server. Allow the fetch size to be set as a server or table option.
10	Push aggregate functions to the remote server, when possible.
11	Allow to push down aggregates to foreign tables that are partitions. Allow to push UPDATEs and DELETEs using joins to foreign servers.
12	Allow ORDER BY sorts and LIMIT clauses to be pushed in more cases.
14	Allow TRUNCATE to operate on foreign tables. Allow INSERT rows in bulk. Add function postgres_fdw_get_connections() to report open foreign server connections.

Since the previous section describes how postgres\_fdw processes a single-table query, the following subsection describes how it processes a multi-table query, sort operation, and aggregate functions.

This subsection focuses on the SELECT statement. However, postgres\_fdw can also process other DML (INSERT, UPDATE, and DELETE) statements as shown ❶ below.

### ❶ PostgreSQL's FDW does not detect deadlock.

The postgres\_fdw extension and the FDW feature do not support the distributed lock manager and the distributed deadlock detection feature. Therefore, a deadlock can easily be generated. For example, if Client\_A updates a local table 'tbl\_local' and a foreign table 'tbl\_remote', and Client\_B updates 'tbl\_remote' and 'tbl\_local', then these two transactions are in deadlock, but PostgreSQL cannot detect it. Therefore, these transactions cannot be committed.

```
localdb=# -- Client A
localdb=# BEGIN;
BEGIN
localdb=# UPDATE tbl_local SET data = 0 WHERE id = 1;
UPDATE 1
localdb=# UPDATE tbl_remote SET data = 0 WHERE id = 1;
UPDATE 1
```

```
localdb=# -- Client B
localdb=# BEGIN;
BEGIN
localdb=# UPDATE tbl_remote SET data = 0 WHERE id = 1;
UPDATE 1
localdb=# UPDATE tbl_local SET data = 0 WHERE id = 1;
UPDATE 1
```

#### 4.1.2.1. Multi-Table Query

To execute a multi-table query, `postgres_fdw` fetches each foreign table using a single-table `SELECT` statement and then join them on the local server.

In versions 9.5 or earlier, even if the foreign tables are stored in the same remote server, `postgres_fdw` fetches them individually and joins them.

In versions 9.6 or later, `postgres_fdw` has been improved and can execute the remote join operation on the remote server when the foreign tables are on the same server and the `use_remote_estimate` option is enabled.

The execution details are described as follows.

##### Versions 9.5 or earlier:

Let us explore how PostgreSQL processes the following query that joins two foreign tables: `tbl_a` and `tbl_b`.

```
localdb=# SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND a.id < 200;
```

The result of the `EXPLAIN` command of the query is shown below:

```
1. localdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND a.id
   < 200;
2.                                     QUERY PLAN
3.
4. Merge Join  (cost=532.31..700.34 rows=10918 width=16)
5.   Merge Cond: (a.id = b.id)
6.   -> Sort  (cost=200.59..202.72 rows=853 width=8)
7.     Sort Key: a.id
8.     -> Foreign Scan on tbl_a a  (cost=100.00..159.06 rows=853 width=8)
9.   -> Sort  (cost=331.72..338.12 rows=2560 width=8)
10.    Sort Key: b.id
11.    -> Foreign Scan on tbl_b b  (cost=100.00..186.80 rows=2560 width=8)
12. (8 rows)
```

The result shows that the executor selects the merge join and is processed as the following steps:

**Line 8:** The executor fetches the table `tbl_a` using the foreign table scan.

**Line 6:** The executor sorts the fetched rows of `tbl_a` on the local server.

**Line 11:** The executor fetches the table `tbl_b` using the foreign table scan.

**Line 9:** The executor sorts the fetched rows of `tbl_b` on the local server.

**Line 4:** The executor carries out a merge join operation on the local server.

The following describes how the executor fetches the rows (Fig. 4.6).

(5-1) Start the remote transaction.

(5-2) Declare the cursor `c1`, the `SELECT` statement of which is shown below:

```
SELECT id,data FROM public.tbl_a WHERE (id < 200)
```

(5-3) Execute `FETCH` commands to obtain the result of the cursor 1.

(5-4) Declare the cursor `c2`, whose `SELECT` statement is shown below:

```
SELECT id,data FROM public.tbl_b
```

Note that the WHERE clause of the original two-table query is "tbl\_a.id = tbl\_b.id AND tbl\_a.id < 200". Therefore, a WHERE clause "tbl\_b.id < 200" can be logically added to the SELECT statement as shown previously. However, postgres\_fdw cannot perform this inference; therefore, the executor has to execute the SELECT statement without any WHERE clauses and has to fetch all rows of the foreign table *tbl\_b*.

This process is inefficient because unnecessary rows must be read from the remote server via the network. Furthermore, the received rows must be sorted to execute the merge join.

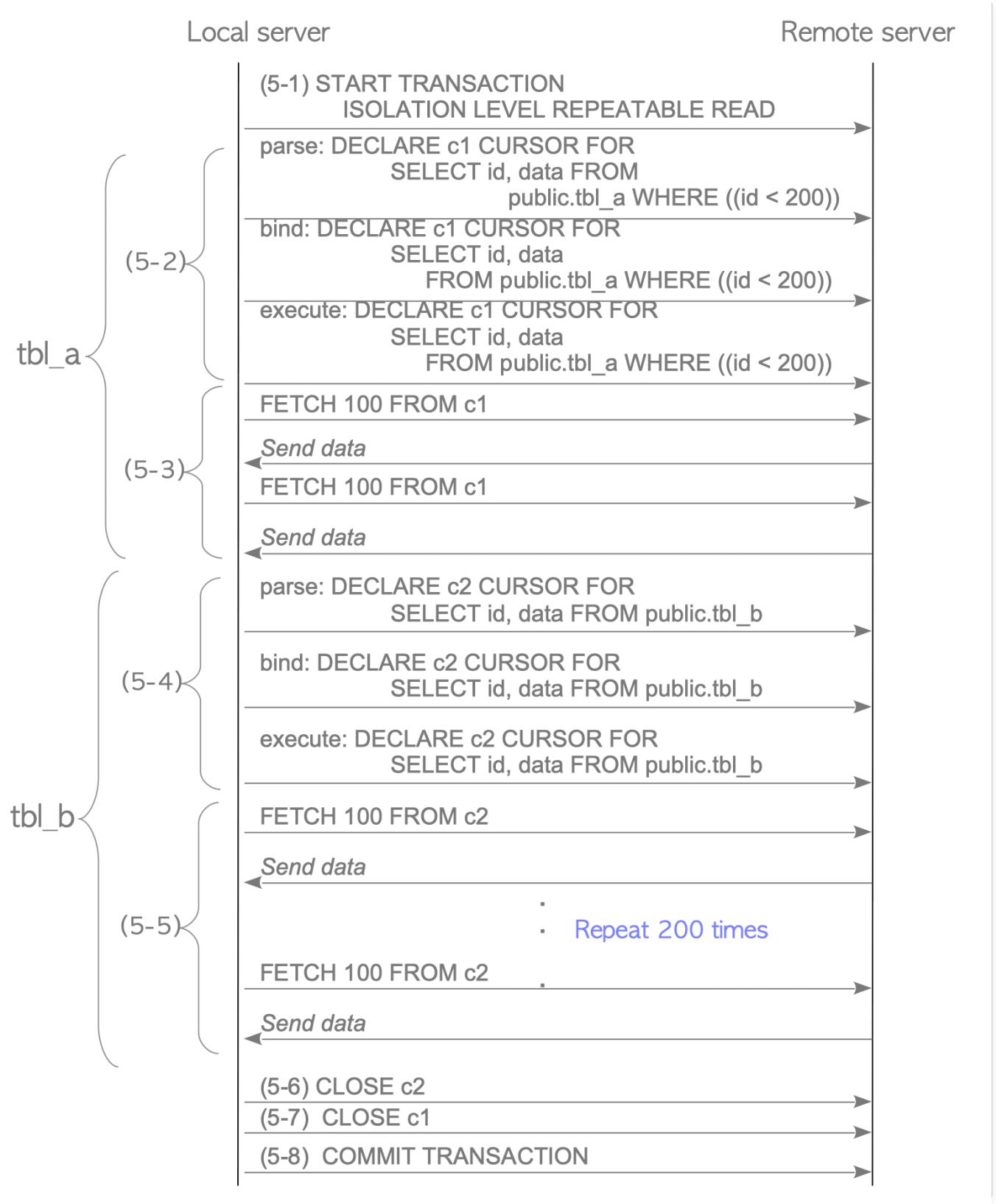
(5-5) Execute FETCH commands to obtain the result of the cursor 2.

(5-6) Close the cursor c1.

(5-7) Close the cursor c2.

(5-8) Commit the transaction.

**Fig. 4.6. Sequence of SQL statements to execute the Multi-Table Query in versions 9.5 or earlier.**



The actual log of the remote server can be found [here](#).

After receiving the rows, the executor sorts both received rows of `tbl_a` and `tbl_b`, and then executes a merge join operation with the sorted rows.

#### Versions 9.6 or later:

If the `use_remote_estimate` option is *on* (the default is *off*), `postgres_fdw` sends several `EXPLAIN` commands to obtain the costs of all plans related to the foreign tables.

To send the EXPLAIN commands, postgres\_fdw sends both the EXPLAIN command of each single-table query and the EXPLAIN commands of the SELECT statements to execute remote join operations. In this example, the following seven EXPLAIN commands are sent to the remote server to obtain the estimated costs of each SELECT statement. The planner then selects the cheapest plan.

```
(1) EXPLAIN SELECT id, data FROM public.tbl_a WHERE ((id < 200))
(2) EXPLAIN SELECT id, data FROM public.tbl_b
(3) EXPLAIN SELECT id, data FROM public.tbl_a WHERE ((id < 200)) ORDER BY id ASC NULLS LAST
(4) EXPLAIN SELECT id, data FROM public.tbl_a WHERE (((SELECT null::integer)::integer = id)) AND ((id < 200))
(5) EXPLAIN SELECT id, data FROM public.tbl_b ORDER BY id ASC NULLS LAST
(6) EXPLAIN SELECT id, data FROM public.tbl_b WHERE (((SELECT null::integer)::integer = id))
(7) EXPLAIN SELECT r1.id, r1.data, r2.id, r2.data FROM (public.tbl_a r1 INNER JOIN public.tbl_b r2 ON ((r1.id = r2.id)) AND ((r1.id < 200)))
```

Let us execute the EXPLAIN command on the local server to observe what plan is selected by the planner.

```
1. localdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND a.id < 200;
2.                                     QUERY PLAN
3. -----
4. Foreign Scan  (cost=134.35..244.45 rows=80 width=16)
5.   Relations: (public.tbl_a a) INNER JOIN (public.tbl_b b)
6.   (2 rows)
```

The result shows that the planner selects the inner join query that is processed on the remote server, which is very efficient.

The following describes how postgres\_fdw is performed (Fig. 4.7).

(3-1) Start the remote transaction.

(3-2) Execute the EXPLAIN commands for estimating the cost of each plan path.

In this example, seven EXPLAIN commands are executed. Then, the planner selects the cheapest cost of the SELECT queries using the results of the executed EXPLAIN commands.

(5-1) Declare the cursor c1, whose SELECT statement is shown below:

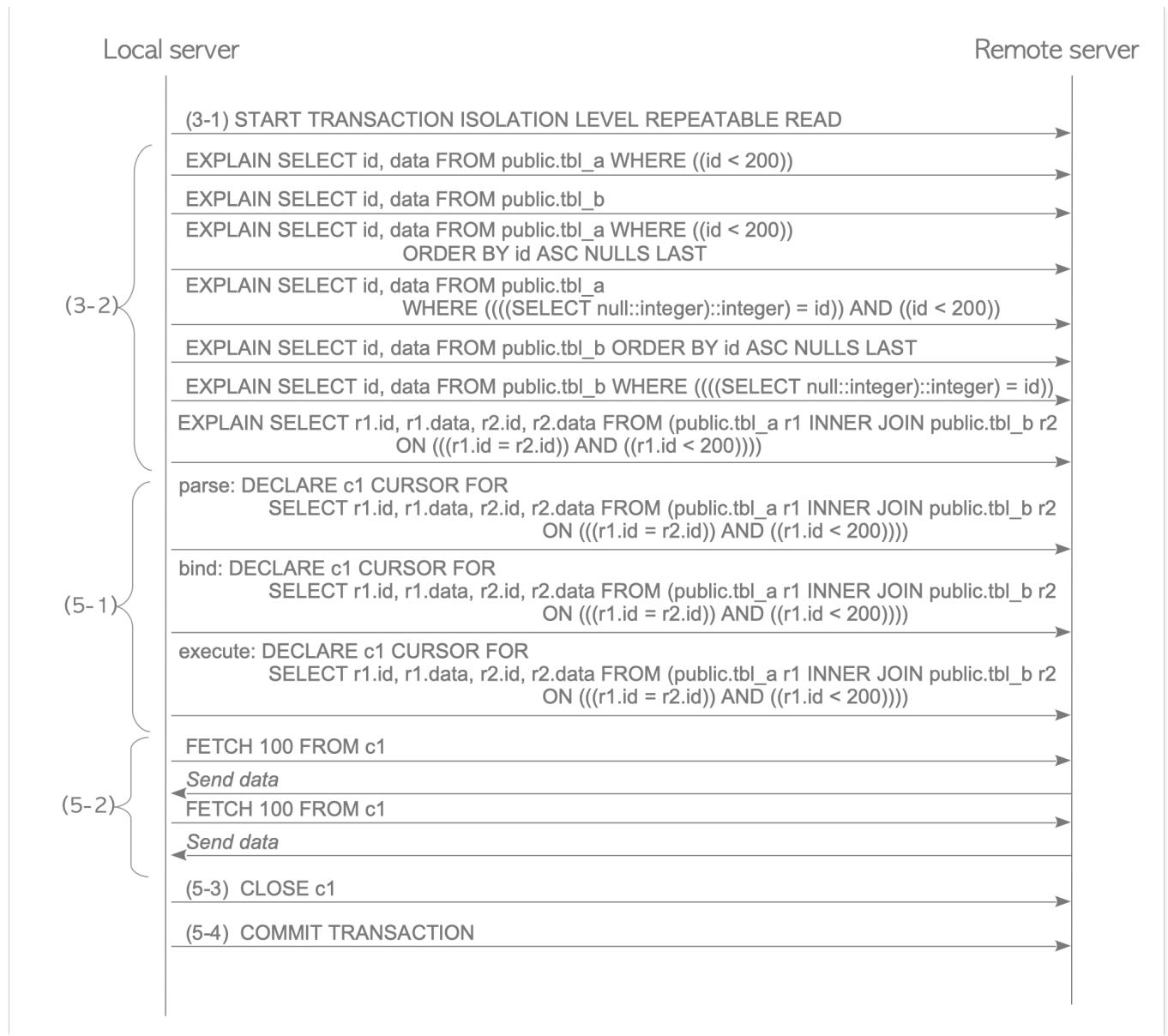
```
SELECT r1.id, r1.data, r2.id, r2.data FROM (public.tbl_a r1 INNER JOIN public.tbl_b r2
ON ((r1.id = r2.id)) AND ((r1.id < 200))))
```

(5-2) Receive the result from the remote server.

(5-3) Close the cursor c1.

(5-4) Commit the transaction.

**Fig. 4.7. Sequence of SQL statements to execute the remote-join operation in versions 9.6 or later.**



The actual log of the remote server can be found [here](#).

Note that if the `use_remote_estimate` option is off (by default), a remote-join query is rarely selected because the costs are estimated using a very large embedded value.

#### 4.1.2.2. Sort Operations

In versions 9.5 or earlier, the sort operation, such as `ORDER BY`, is processed on the local server. This means that the local server fetches all the target rows from the remote server before the sort operation. Let us explore how a simple query that includes an `ORDER BY` clause is processed using the `EXPLAIN` command.

```

1. localdb=# EXPLAIN SELECT * FROM tbl_a AS a WHERE a.id < 200 ORDER BY a.id;
2.                                     QUERY PLAN
3. -----
4.   Sort  (cost=200.59..202.72 rows=853 width=8)
5.     Sort Key: id
6.     -> Foreign Scan on tbl_a a  (cost=100.00..159.06 rows=853 width=8)
7.   (3 rows)

```

**Line 6:** The executor sends the following query to the remote server, and then fetches the query result.

```
SELECT id, data FROM public.tbl_a WHERE ((id < 200))
```

**Line 4:** The executor sorts the fetched rows of `tbl_a` on the local server.

The actual log of the remote server can be found [here](#).

In versions 9.6 or later, `postgres_fdw` can execute the `SELECT` statements with an `ORDER BY` clause on the remote server if possible.

```
1. localdb=# EXPLAIN SELECT * FROM tbl_a AS a WHERE a.id < 200 ORDER BY a.id;
   QUERY PLAN
   -----
4. Foreign Scan on tbl_a a  (cost=100.00..167.46 rows=853 width=8)
5. (1 row)
```

**Line 4:** The executor sends the following query that has an `ORDER BY` clause to the remote server, and then fetches the query result, which is already sorted.

```
SELECT id, data FROM public.tbl_a WHERE ((id < 200)) ORDER BY id ASC NULLS LAST
```

The actual log of the remote server can be found [here](#). This improvement has reduced the workload of the local server.

#### 4.1.2.3. Aggregate Functions

In versions 9.6 or earlier, similar to the sort operation mentioned in the previous subsection, aggregate functions such as `AVG()` and `COUNT()` are processed on the local server as the following steps:

```
1. localdb=# EXPLAIN SELECT AVG(data) FROM tbl_a AS a WHERE a.id < 200;
   QUERY PLAN
   -----
4. Aggregate  (cost=168.50..168.51 rows=1 width=4)
5.    -> Foreign Scan on tbl_a a  (cost=100.00..166.06 rows=975 width=4)
6. (2 rows)
```

**Line 5:** The executor sends the following query to the remote server, and then fetches the query result.

```
SELECT id, data FROM public.tbl_a WHERE ((id < 200))
```

**Line 4:** The executor computes the average of the fetched rows of `tbl_a` on the local server.

The actual log of the remote server can be found [here](#). This process is costly because sending a large number of rows consumes heavy network traffic and takes a long time.

In versions 10 or later, `postgres_fdw` executes the `SELECT` statement with the aggregate function on the remote server if possible.

```
1. localdb=# EXPLAIN SELECT AVG(data) FROM tbl_a AS a WHERE a.id < 200;
   QUERY PLAN
   -----
4. Foreign Scan  (cost=102.44..149.03 rows=1 width=32)
5.    Relations: Aggregate on (public.tbl_a a)
6. (2 rows)
```

**Line 4:** The executor sends the following query that contains an `AVG()` function to the remote server, and then fetches the query result.

```
SELECT avg(data) FROM public.tbl_a WHERE ((id < 200))
```

The actual log of the remote server can be found here. This process is obviously efficient because the remote server calculates the average and sends only one row as the result.

### ⓘ Push-down

Similar to the given example, **push-down** is an operation where the local server allows the remote server to process some operations, such as aggregate procedures.

## 4.2. Parallel Query



# Chapter 5

## Concurrency Control

---

**C**oncurrency Control is a mechanism that maintains atomicity and isolation, which are two properties of the ACID, when multiple transactions run concurrently in a database.

There are three broad concurrency control techniques: *Multi-version Concurrency Control* (MVCC), *Strict Two-Phase Locking* (S2PL), and *Optimistic Concurrency Control* (OCC). Each technique has many variations. In MVCC, each write operation creates a new version of a data item while retaining the old version. When a transaction reads a data item, the system selects one of the versions to ensure isolation of the individual transaction. The main advantage of MVCC is that '*readers don't block writers, and writers don't block readers*', in contrast, for example, an S2PL-based system must block readers when a writer writes an item because the writer acquires an exclusive lock for the item. PostgreSQL and some RDBMSs use a variation of MVCC called **Snapshot Isolation (SI)**.

To implement SI, some RDBMSs, such as Oracle, use rollback segments. When writing a new data item, the old version of the item is written to the rollback segment, and subsequently the new item is overwritten to the data area. PostgreSQL uses a simpler method. A new data item is inserted directly into the relevant table page. When reading items, PostgreSQL selects the appropriate version of an item in response to an individual transaction by applying **visibility check rules**.

SI does not allow the three anomalies defined in the ANSI SQL-92 standard: *Dirty Reads*, *Non-Repeatable Reads*, and *Phantom Reads*. However, SI cannot achieve true serializability because it allows serialization anomalies, such as *Write Skew* and *Read-only Transaction Skew*. Note that the ANSI SQL-92 standard based on the classical serializability definition is **not** equivalent to the definition in modern theory. To deal with this issue, **Serializable Snapshot Isolation (SSI)** has been added as of version 9.1. SSI can detect the serialization anomalies and can resolve the conflicts caused by such anomalies. Thus, PostgreSQL versions 9.1 or later provide a true **SERIALIZABLE** isolation level. (In addition, SQL Server also uses SSI; Oracle still uses only SI.)

This chapter comprises the following four parts:

**Part 1:** Sections 5.1. — 5.3.

This part provides basic information required for understanding the subsequent parts.

Sections 5.1 and 5.2 describe transaction ids and tuple structure, respectively. Section 5.3 exhibits how tuples are inserted, deleted, and updated.

**Part 2:** Sections 5.4. — 5.6.

This part illustrates the key features required for implementing the concurrency control mechanism.

Sections 5.4, 5.5, and 5.6 describe the commit log (clog), which holds all transaction states, transaction snapshots, and the visibility check rules, respectively.

**Part 3:** Sections 5.7. — 5.9.

This part describes the concurrency control in PostgreSQL using specific examples.

Section 5.7 describes the visibility check. This section also shows how the three anomalies defined in the ANSI SQL standard are prevented. Section 5.8 describes preventing *Lost Updates*, and Section 5.9 briefly describes SSI.

#### Part 4: Section 5.10.

This part describes several maintenance process required to permanently running the concurrency control mechanism. The maintenance processes are performed by vacuum processing, which is described in Chapter 6.

This chapter focuses on the topics that are unique to PostgreSQL, although there are many concurrency control-related topics. Note that descriptions of deadlock prevention and lock modes are omitted. (For more information, refer to the official documentation.)

### 1 Transaction Isolation Level in PostgreSQL

PostgreSQL-implemented transaction isolation levels are described in the following table:

Isolation Level	Dirty Reads	Non-repeatable Read	Phantom Read	Serialization Anomaly
READ COMMITTED	Not possible	Possible	Possible	Possible
REPEATABLE READ <sup>*1</sup>	Not possible	Not possible	Not possible in PG; See Section 5.7.2. (Possible in ANSI SQL)	Possible
SERIALIZABLE	Not possible	Not possible	Not possible	Not possible

\*1 : In versions 9.0 and earlier, this level had been used as 'SERIALIZABLE' because it does not allow the three anomalies defined in the ANSI SQL-92 standard. However, with the implementation of SSI in version 9.1, this level has changed to 'REPEATABLE READ' and a true SERIALIZABLE level was introduced.



PostgreSQL uses SSI for DML (Data Manipulation Language, e.g., SELECT, UPDATE, INSERT, DELETE), and 2PL for DDL (Data Definition Language, e.g., CREATE TABLE, etc).

## 5.1. Transaction ID

Whenever a transaction begins, a unique identifier, referred to as a **transaction id (txid)**, is assigned by the transaction manager. PostgreSQL's txid is a 32-bit unsigned integer, approximately 4.2 billion (thousand millions). If you execute the built-in *txid\_current()* function after a transaction starts, the function returns the current txid as follows:

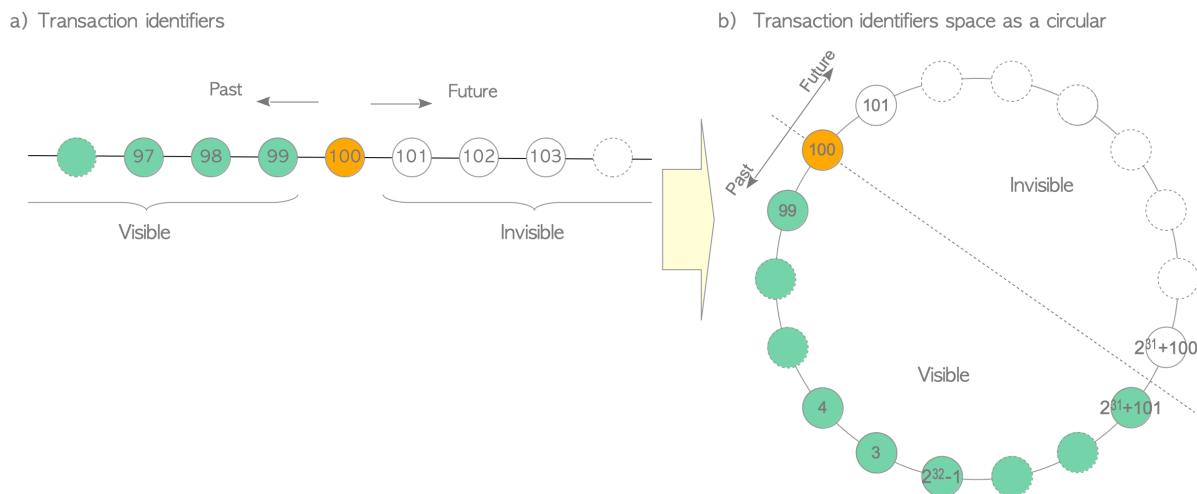
```
testdb=# BEGIN;
BEGIN
testdb=# SELECT txid_current();
 txid_current
-----
```

PostgreSQL reserves the following three special txids:

- 0** means **Invalid** txid.
- 1** means **Bootstrap** txid, which is only used in the initialization of the database cluster.
- 2** means **Frozen** txid, which is described in Section 5.10.1.

Txids can be compared with each other. For example, at the viewpoint of txid 100, txids that are greater than 100 are '*in the future*' and are *invisible* from the txid 100; txids that are less than 100 are '*in the past*' and are *visible* (Fig. 5.1 a)).

**Fig. 5.1. Transaction ids in PostgreSQL.**



Since the txid space is insufficient in practical systems, PostgreSQL treats the txid space as a circle. The previous 2.1 billion txids are '*in the past*', and the next 2.1 billion txids are '*in the future*' (Fig. 5.1 b)).

Note that the so-called *txid wraparound problem* is described in Section 5.10.1.



Note that BEGIN command does not be assigned a txid. In PostgreSQL, when the first command is executed after a BEGIN command executed, a txid is assigned by the transaction manager, and then the transaction starts.

## 5.2. Tuple Structure

Heap tuples in table pages are classified into two types: usual data tuples and TOAST tuples. This section describes only the usual tuple.

A heap tuple comprises three parts: the HeapTupleHeaderData structure, NULL bitmap, and user data (Fig. 5.2).

**Fig. 5.2. Tuple structure.**



The `HeapTupleHeaderData` structure is defined in `src/include/access/htup_details.h`.

The `HeapTupleHeaderData` structure contains seven fields, but only four of them are required in the subsequent sections:

- **`t_xmin`** holds the txid of the transaction that inserted this tuple.
- **`t_xmax`** holds the txid of the transaction that deleted or updated this tuple. If this tuple has not been deleted or updated, `t_xmax` is set to 0, which means INVALID.
- **`t_cid`** holds the command id (cid), which is the number of SQL commands that were executed before this command was executed within the current transaction, starting from 0. For example, assume that we execute three INSERT commands within a single transaction: 'BEGIN; INSERT; INSERT; COMMIT;'. If the first command inserts this tuple, `t_cid` is set to 0. If the second command inserts this tuple, `t_cid` is set to 1, and so on.
- **`t_ctid`** holds the tuple identifier (tid) that points to itself or a new tuple. `tid`, described in Section 1.3, is used to identify a tuple within a table. When this tuple is updated, the `t_ctid` of this tuple points to the new tuple; otherwise, the `t_ctid` points to itself.

```

typedef struct HeapTupleFields
{
    TransactionId t_xmin;           /* inserting xact ID */
    TransactionId t_xmax;           /* deleting or locking xact ID */

    union
    {
        CommandId      t_cid;       /* inserting or deleting command ID, or
both */
        TransactionId   t_xvac;     /* old-style VACUUM FULL xact ID
*/
    } t_field3;
} HeapTupleFields;

typedef struct DatumTupleFields
{
    int32          datum_len_;      /* varlena header (do not touch directl
y!) */
    int32          datum_tymod;     /* -1, or identifier of a record type */
    Oid            datum_typeid;   /* composite type OID, or RECORDOID */

    /*
     * Note: field ordering is chosen with thought that Oid might someday
     * widen to 64 bits.
    */
} DatumTupleFields;

```

```

typedef struct HeapTupleHeaderData
{
    union
    {
        HeapTupleFields t_heap;
        DatumTupleFields t_datum;
    } t_choice;

    ItemPointerData t_ctid;           /* current TID of this or newer tuple */

    /* Fields below here must match MinimalTupleData! */
    uint16          t_infomask2;     /* number of attributes + various flags */
    uint16          t_infomask;      /* various flag bits, see below */
    uint8           t_hoff;         /* sizeof header incl. bitmap, padding */
    /* ^ - 23 bytes - ^ */
    bits8          t_bits[1];       /* bitmap of NULLs -- VARIABLE LENGTH */

    /* MORE DATA FOLLOWS AT END OF STRUCT */
} HeapTupleHeaderData;

typedef HeapTupleHeaderData *HeapTupleHeader;

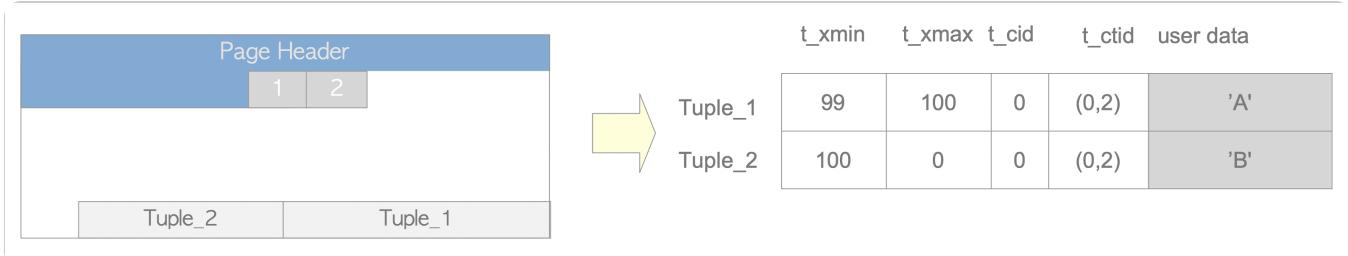
```

## 5.3. Inserting, Deleting and Updating Tuples

This section describes how tuples are inserted, deleted, and updated. Then, the *Free Space Map (FSM)*, which is used to insert and update tuples, is briefly described.

To focus on tuples, page headers and line pointers are not represented in the following. Figure 5.3 shows an example of how tuples are represented.

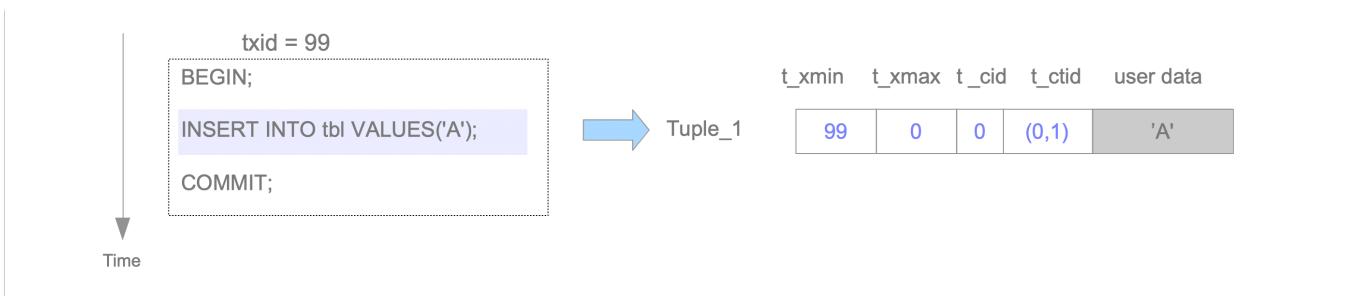
**Fig. 5.3. Representation of tuples.**



### 5.3.1. Insertion

With the insertion operation, a new tuple is inserted directly into a page of the target table (Fig. 5.4).

**Fig. 5.4. Tuple insertion.**



Suppose that a tuple is inserted in a page by a transaction whose txid is 99. In this case, the header fields of the inserted tuple are set as follows.

Tuple\_1:

- t\_xmin** is set to 99 because this tuple is inserted by txid 99.
- t\_xmax** is set to 0 because this tuple has not been deleted or updated.
- t\_cid** is set to 0 because this tuple is the first tuple inserted by txid 99.
- t\_ctid** is set to (0,1), which points to itself, because this is the latest tuple.

## 🎓 pageinspect

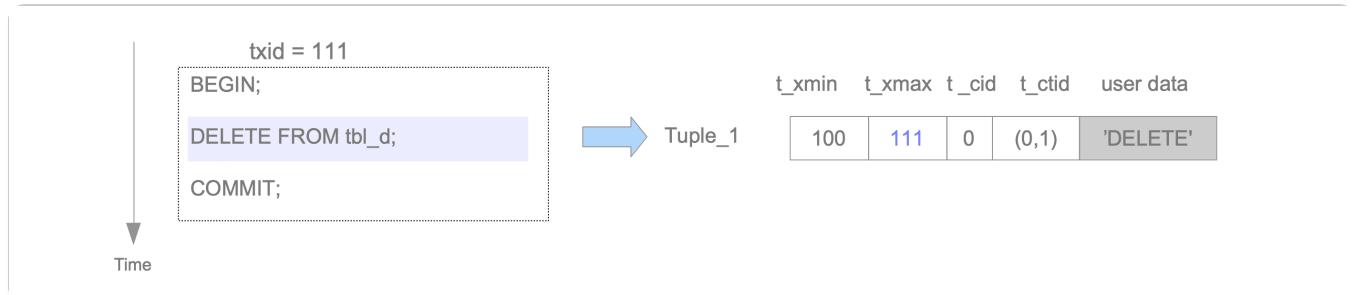
PostgreSQL provides an extension *pageinspect*, which is a contribution module, to show the contents of the database pages.

```
testdb=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
testdb=# CREATE TABLE tbl (data text);
CREATE TABLE
testdb=# INSERT INTO tbl VALUES('A');
INSERT 0 1
testdb=# SELECT lp AS tuple, t_xmin, t_xmax, t_field3 AS t_cid, t_ctid
          FROM heap_page_items(get_raw_page('tbl', 0));
tuple | t_xmin | t_xmax | t_cid | t_ctid
-----+-----+-----+-----+
      1 |    99 |      0 |     0 | (0,1)
(1 row)
```

### 5.3.2. Deletion

In the deletion operation, the target tuple is deleted logically. The value of the txid that executes the DELETE command is set to the **t\_xmax** of the tuple (Fig. 5.5).

**Fig. 5.5. Tuple deletion.**



Suppose that tuple Tuple\_1 is deleted by txid 111. In this case, the header fields of Tuple\_1 are set as follows:

Tuple\_1:

- t\_xmax** is set to 111.

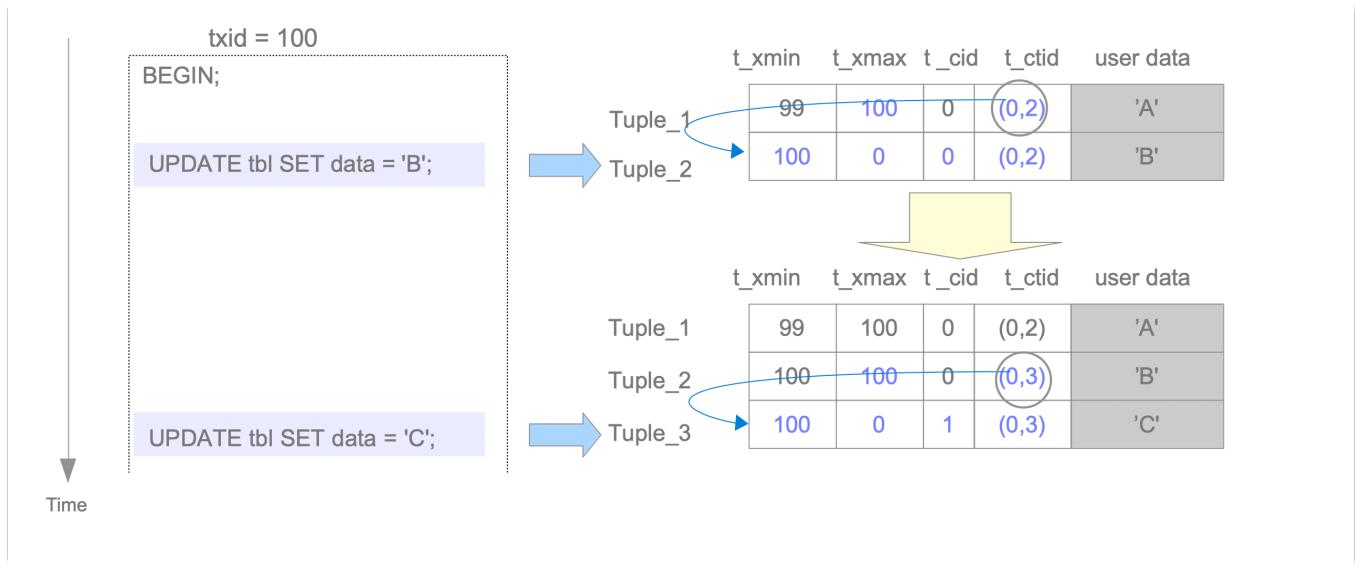
If txid 111 is committed, Tuple\_1 is no longer required. Generally, unneeded tuples are referred to as **dead tuples** in PostgreSQL.

Dead tuples should eventually be removed from pages. Cleaning dead tuples is referred to as **VACUUM** processing, which is described in Chapter 6.

### 5.3.3. Update

In the update operation, PostgreSQL logically deletes the latest tuple and inserts a new one (Fig. 5.6).

**Fig. 5.6. Update the row twice.**



Suppose that the row, which has been inserted by txid 99, is updated twice by txid 100.

When the first `UPDATE` command is executed, Tuple\_1 is logically deleted by setting `t_xid` 100 to the `t_xmax`, and then Tuple\_2 is inserted. Then, the `t_ctid` of Tuple\_1 is rewritten to point to Tuple\_2. The header fields of both Tuple\_1 and Tuple\_2 are as follows:

Tuple\_1:

- t\_xmax** is set to 100.
- t\_ctid** is rewritten from (0, 1) to (0, 2).

Tuple\_2:

- t\_xmin** is set to 100.
- t\_xmax** is set to 0.
- t\_cid** is set to 0.
- t\_ctid** is set to (0,2).

When the second `UPDATE` command is executed, as in the first `UPDATE` command, Tuple\_2 is logically deleted and Tuple\_3 is inserted. The header fields of both Tuple\_2 and Tuple\_3 are as follows:

Tuple\_2:

- t\_xmax** is set to 100.
- t\_ctid** is rewritten from (0, 2) to (0, 3).

Tuple\_3:

- t\_xmin** is set to 100.
- t\_xmax** is set to 0.
- t\_cid** is set to 1.
- t\_ctid** is set to (0,3).

As with the delete operation, if txid 100 is committed, Tuple\_1 and Tuple\_2 will be dead tuples, and, if txid 100 is aborted, Tuple\_2 and Tuple\_3 will be dead tuples.

## 5.3.4. Free Space Map

When inserting a heap or an index tuple, PostgreSQL uses the **FSM** of the corresponding table or index to select the page which can be inserted into.

As mentioned in Section 1.2.3, all tables and indexes have respective FSMs. Each FSM stores the information about the free space capacity of each page within the corresponding table or index file.

All FSMs are stored with the suffix 'fsm', and they are loaded into shared memory if necessary.

### pg\_freespacemap

The extension `pg_freespacemap` provides the freespace of the specified table/index. The following query shows the freespace ratio of each page in the specified table.

```
testdb=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION

testdb=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
          FROM pg_freespace('accounts');
blkno | avail | freespace ratio
-----+-----+-----
  0  | 7904  |      96.00
  1  | 7520  |      91.00
  2  | 7136  |      87.00
  3  | 7136  |      87.00
  4  | 7136  |      87.00
  5  | 7136  |      87.00
....
```

## 5.4. Commit Log (clog)

PostgreSQL holds the statuses of transactions in the **Commit Log**. The Commit Log, often called the **clog**, is allocated to shared memory and is used throughout transaction processing.

This section describes the the status of transactions in PostgreSQL, how the clog operates, and maintenance of the clog.

### 5.4.1 Transaction Status

PostgreSQL defines four transaction states: IN\_PROGRESS, COMMITTED, ABORTED, and SUB\_COMMITTED.

The first three statuses are self-explanatory. For example, when a transaction is in progress, its status is IN\_PROGRESS.

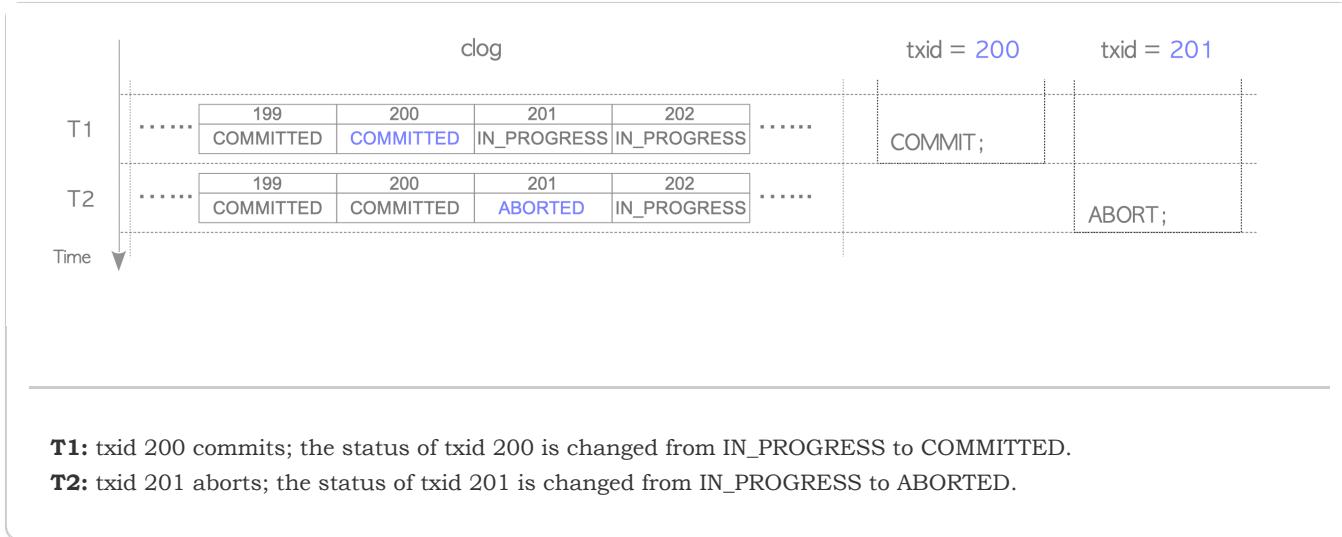
SUB\_COMMITTED is for sub-transactions, and its description is omitted in this document.

### 5.4.2. How Clog Performs

The clog comprises one or more 8 KB pages in shared memory. It logically forms an array, where the indices of the array correspond to the respective transaction ids, and each item in the array

holds the status of the corresponding transaction id. Figure 5.7 shows the clog and how it operates.

**Fig. 5.7. How the clog operates.**



When the current txid advances and the clog can no longer store it, a new page is appended.

When the status of a transaction is needed, internal functions are invoked. Those functions read the clog and return the status of the requested transaction. (See also ['Hint Bits'](#) in Section 5.7.1.)

### 5.4.3. Maintenance of the Clog

When PostgreSQL shuts down or whenever the checkpoint process runs, the data of the clog are written into files stored in the `pg_xact` subdirectory. (Note that `pg_xact` was called `pg_clog` in versions 9.6 or earlier.) These files are named `0000`, `0001`, and so on. The maximum file size is 256 KB. For example, if the clog uses eight pages (the first page to the eighth page, the total size is 64 KB), its data are written into `0000` (64 KB). If the clog uses 37 pages (296 KB), its data are written into `0000` and `0001`, which are 256 KB and 40 KB in size, respectively.

When PostgreSQL starts up, the data stored in the `pg_xact` files are loaded to initialize the clog.

The size of the clog continuously increases because a new page is appended whenever the clog is filled up. However, not all data in the clog are necessary. Vacuum processing, described in Chapter 6, regularly removes such old data (both the clog pages and files). Details about removing the clog data are described in Section 6.4.

## 5.5. Transaction Snapshot

A **transaction snapshot** is a dataset that stores information about whether all transactions are active at a certain point in time for an individual transaction. Here an active transaction means it is in progress or has not yet started.

PostgreSQL internally defines the textual representation format of transaction snapshots as '`100:100:`'. For example, '`100:100:`' means 'txids that are less than 99 are not active, and txids that are equal or greater than 100 are active'. In the following descriptions, this convenient representation form is used. If you are not familiar with it, see ['Hint Bits'](#) below.

## ❶ The built-in function pg\_current\_snapshot and its textual representation format

The function pg\_current\_snapshot shows a snapshot of the current transaction.

```
testdb=# SELECT pg_current_snapshot();  
pg_current_snapshot  
-----  
100:104:100,102  
(1 row)
```

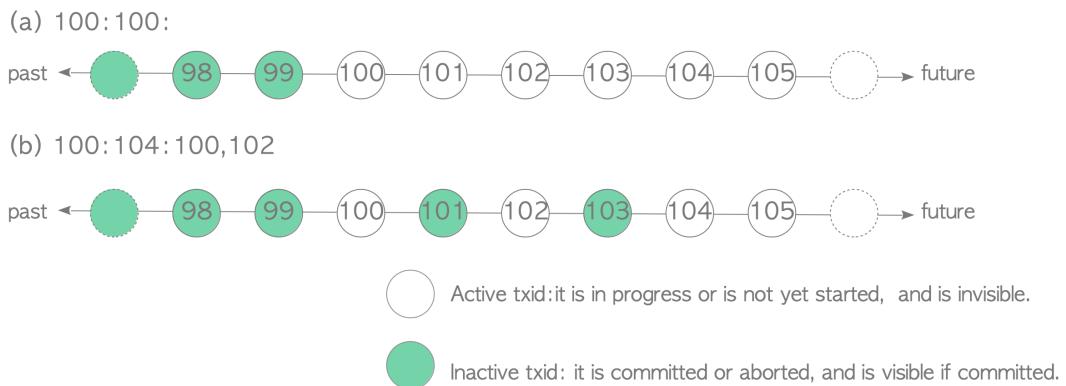
The textual representation of the txid\_current\_snapshot is '**xmin:xmax:xip\_list**', and the components are described as follows:

- **xmin**  
(earliest txid that is still active): All earlier transactions will either be committed and visible, or rolled back and dead.
- **xmax**  
(first as-yet-unassigned txid): All txids greater than or equal to this are not yet started as of the time of the snapshot, and thus invisible.
- **xip\_list**  
(list of active transaction ids at the time of the snapshot): The list includes only active txids between xmin and xmax.

For example, in the snapshot '100:104:100,102', xmin is '100', xmax '104', and xip\_list '100,102'.

Here are two specific examples:

**Fig. 5.8. Examples of transaction snapshot representation.**



The first example is '**100:100:**'. This snapshot means the following (Fig. 5.8(a)):

- txids equal or less than 99 are **not active** because xmin is 100.
- txids equal or greater than 100 are **active** because xmax is 100.

The second example is '**100:104:100,102**'. This snapshot means the following (Fig. 5.8(b)):

- txids equal or less than 99 are **not active**.
- txids equal or greater than 104 are **active**.
- txids 100 and 102 are **active** since they exist in the xip list, whereas txids 101 and 103 are **not active**.

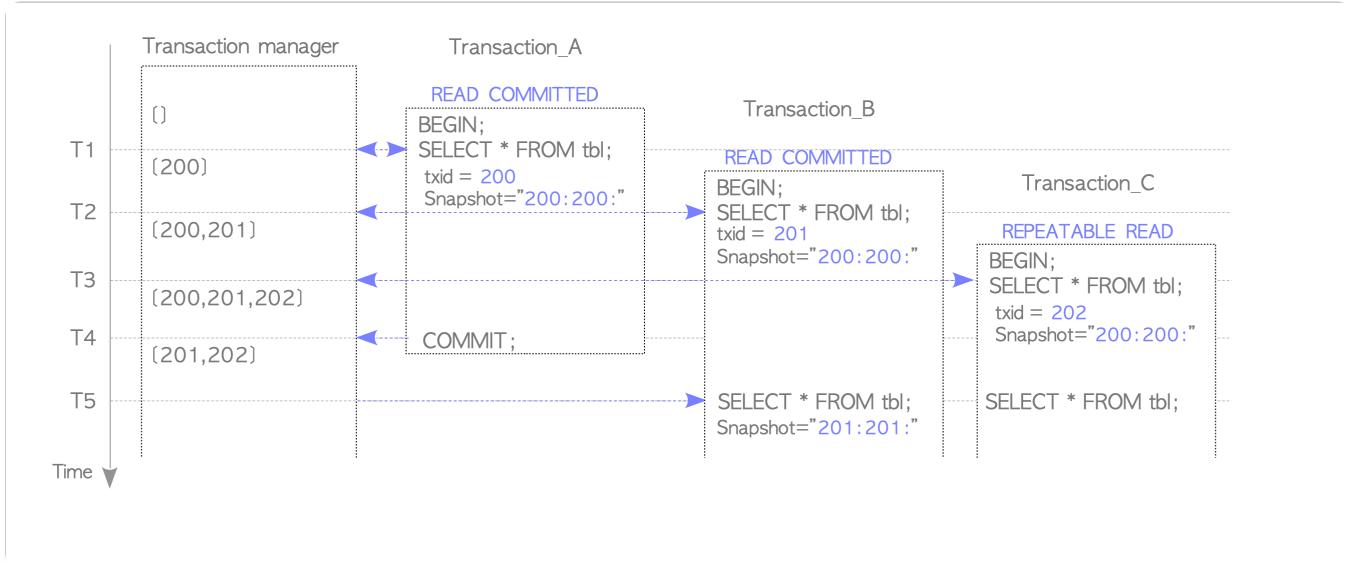
Transaction snapshots are provided by the transaction manager. In the READ COMMITTED isolation level, the transaction obtains a snapshot whenever an SQL command is executed;

otherwise (REPEATABLE READ or SERIALIZABLE), the transaction only gets a snapshot when the first SQL command is executed. The obtained transaction snapshot is used for a visibility check of tuples, which is described in Section 5.7.

When using the obtained snapshot for the visibility check, *active* transactions in the snapshot must be treated as *in progress* even if they have actually been committed or aborted. This rule is important because it causes the difference in the behavior between READ COMMITTED and REPEATABLE READ (or SERIALIZABLE). We refer to this rule repeatedly in the following sections.

In the remainder of this section, the transaction manager and transactions are described using a specific scenario Fig. 5.9.

**Fig. 5.9. Transaction manager and transactions.**



The transaction manager always holds information about currently running transactions. Suppose that three transactions start one after another, and the isolation level of Transaction\_A and Transaction\_B are READ COMMITTED, and that of Transaction\_C is REPEATABLE READ.

#### T1:

Transaction\_A starts and executes the first SELECT command. When executing the first command, Transaction\_A requests the txid and snapshot of this moment. In this scenario, the transaction manager assigns txid 200, and returns the transaction snapshot '200:200:'.

#### T2:

Transaction\_B starts and executes the first SELECT command. The transaction manager assigns txid 201, and returns the transaction snapshot '200:200:' because Transaction\_A (txid 200) is in progress. Thus, Transaction\_A cannot be seen from Transaction\_B.

#### T3:

Transaction\_C starts and executes the first SELECT command. The transaction manager assigns txid 202, and returns the transaction snapshot '200:200:', thus, Transaction\_A and Transaction\_B cannot be seen from Transaction\_C.

#### T4:

Transaction\_A has been committed. The transaction manager removes the information about this transaction.

#### T5:

Transaction\_B and Transaction\_C execute their respective SELECT commands.

Transaction\_B requires a transaction snapshot because it is in the READ COMMITTED level. In this scenario, Transaction\_B obtains a new snapshot '201:201:' because Transaction\_A (txid

200) is committed. Thus, Transaction\_A is no longer invisible from Transaction\_B.

Transaction\_C does not require a transaction snapshot because it is in the REPEATABLE READ level and uses the obtained snapshot, i.e. '200:200:'. Thus, Transaction\_A is still invisible from Transaction\_C.

## 5.6. Visibility Check Rules

Visibility check rules are a set of rules used to determine whether each tuple is visible or invisible using both the t\_xmin and t\_xmax of the tuple, the clog, and the obtained transaction snapshot. These rules are too complicated to explain in detail. Therefore this document shows the minimal rules required for the subsequent descriptions. In the following, we omit the rules related to sub-transactions and ignore discussion about t\_ctid, i.e. we do not consider tuples that have been updated more than twice within a transaction.

The number of selected rules is ten, and they can be classified into three cases.

### 5.6.1. Status of t\_xmin is ABORTED

A tuple whose t\_xmin status is ABORTED is always *invisible* (Rule 1) because the transaction that inserted this tuple has been aborted.

```
/* t_xmin status == ABORTED */
Rule 1: IF t_xmin status is 'ABORTED' THEN
    RETURN 'Invisible'
END IF
```

This rule is explicitly expressed as the following mathematical expression.

**Rule 1:** If Status(t\_xmin) = ABORTED  $\Rightarrow$  Invisible

### 5.6.2. Status of t\_xmin is IN\_PROGRESS

A tuple whose t\_xmin status is IN\_PROGRESS is essentially *invisible* (Rules 3 and 4), except under one condition.

```
/* t_xmin status == IN_PROGRESS */
IF t_xmin status is 'IN_PROGRESS' THEN
    IF t_xmin = current_txid THEN
        IF t_xmax = INVALID THEN
            RETURN 'Visible'
        ELSE /* this tuple has been deleted or updated by the current transaction itself. */
            RETURN 'Invisible'
        END IF
    ELSE /* t_xmin != current_txid */
        RETURN 'Invisible'
    END IF
END IF
```

If this tuple is inserted by another transaction and the status of t\_xmin is IN\_PROGRESS, then this tuple is obviously *invisible* (Rule 4).

If t\_xmin is equal to the current txid (i.e., this tuple is inserted by the current transaction) and t\_xmax is **not** INVALID, then this tuple is *invisible* because it has been updated or deleted by the current transaction (Rule 3).

The exception condition is the case where this tuple is inserted by the current transaction and t\_xmax is INVALID. In this case, this tuple must be *visible* from the current transaction (Rule 2)

because this tuple is the tuple inserted by the current transaction itself.

**Rule 2:** If  $\text{Status}(t_{\text{xmin}}) = \text{IN\_PROGRESS} \wedge t_{\text{xmin}} = \text{current\_txid} \wedge t_{\text{xmax}} = \text{INVALID} \Rightarrow \text{Visible}$

**Rule 3:** If  $\text{Status}(t_{\text{xmin}}) = \text{IN\_PROGRESS} \wedge t_{\text{xmin}} = \text{current\_txid} \wedge t_{\text{xmax}} \neq \text{INVALID} \Rightarrow \text{Invisible}$

**Rule 4:** If  $\text{Status}(t_{\text{xmin}}) = \text{IN\_PROGRESS} \wedge t_{\text{xmin}} \neq \text{current\_txid} \Rightarrow \text{Invisible}$

### 5.6.3. Status of $t_{\text{xmin}}$ is COMMITTED

A tuple whose  $t_{\text{xmin}}$  status is COMMITTED is *visible* (Rules 6,8, and 9), except under three conditions.

```
/*  $t_{\text{xmin}}$  status == COMMITTED */
  IF  $t_{\text{xmin}}$  status is 'COMMITTED' THEN
    Rule 5:  IF  $t_{\text{xmin}}$  is active in the obtained transaction snapshot THEN
              RETURN 'Invisible'
    Rule 6:  ELSE IF  $t_{\text{xmax}} = \text{INVALID}$  OR status of  $t_{\text{xmax}}$  is 'ABORTED' THEN
              RETURN 'Visible'
            ELSE IF  $t_{\text{xmax}}$  status is 'IN_PROGRESS' THEN
    Rule 7:      IF  $t_{\text{xmax}} = \text{current\_txid}$  THEN
                  RETURN 'Invisible'
    Rule 8:      ELSE /*  $t_{\text{xmax}} \neq \text{current\_txid}$  */
                  RETURN 'Visible'
                END IF
            ELSE IF  $t_{\text{xmax}}$  status is 'COMMITTED' THEN
    Rule 9:      IF  $t_{\text{xmax}}$  is active in the obtained transaction snapshot THEN
                  RETURN 'Visible'
    Rule 10:    ELSE
                  RETURN 'Invisible'
                END IF
              END IF
            END IF
  END IF
```

Rule 6 is obvious because  $t_{\text{xmax}}$  is INVALID or ABORTED. Three exception conditions and both Rules 8 and 9 are described as follows:

The first exception condition is that  $t_{\text{xmin}}$  is *active* in the obtained transaction snapshot (Rule 5). Under this condition, this tuple is *invisible* because  $t_{\text{xmin}}$  should be treated as in progress.

The second exception condition is that  $t_{\text{xmax}}$  is the current txid (Rule 7). Under this condition, as with Rule 3, this tuple is *invisible* because it has been updated or deleted by this transaction itself.

In contrast, if the status of  $t_{\text{xmax}}$  is IN\_PROGRESS and  $t_{\text{xmax}}$  is not the current txid (Rule 8), the tuple is *visible* because it has not been deleted.

The third exception condition is that the status of  $t_{\text{xmax}}$  is COMMITTED and  $t_{\text{xmax}}$  is **not** active in the obtained transaction snapshot (Rule 10). Under this condition, this tuple is *invisible* because it has been updated or deleted by another transaction.

In contrast, if the status of  $t_{\text{xmax}}$  is COMMITTED but  $t_{\text{xmax}}$  is active in the obtained transaction snapshot (Rule 9), the tuple is *visible* because  $t_{\text{xmax}}$  should be treated as in progress.

**Rule 5:** If  $\text{Status}(t_{\text{xmin}}) = \text{COMMITTED} \wedge \text{Snapshot}(t_{\text{xmin}}) = \text{active} \Rightarrow \text{Invisible}$

**Rule 6:** If  $\text{Status}(t_{\text{xmin}}) = \text{COMMITTED} \wedge (t_{\text{xmax}} = \text{INVALID} \vee \text{Status}(t_{\text{xmax}}) = \text{ABORTED}) \Rightarrow \text{Visible}$

**Rule 7:** If  $\text{Status}(t_{\text{xmin}}) = \text{COMMITTED} \wedge \text{Status}(t_{\text{xmax}}) = \text{IN\_PROGRESS} \wedge t_{\text{xmax}} = \text{current\_txid} \Rightarrow \text{Invisible}$

**Rule 8:** If  $\text{Status}(t_{\text{xmin}}) = \text{COMMITTED} \wedge \text{Status}(t_{\text{xmax}}) = \text{IN\_PROGRESS} \wedge t_{\text{xmax}} \neq \text{current\_txid} \Rightarrow \text{Visible}$

**Rule 9:** If  $\text{Status}(t_{\text{xmin}}) = \text{COMMITTED} \wedge \text{Status}(t_{\text{xmax}}) = \text{COMMITTED} \wedge \text{Snapshot}(t_{\text{xmax}}) = \text{active} \Rightarrow \text{Visible}$

**Rule 10:** If  $\text{Status}(t_{\text{xmin}}) = \text{COMMITTED} \wedge \text{Status}(t_{\text{xmax}}) = \text{COMMITTED} \wedge \text{Snapshot}(t_{\text{xmax}}) \neq \text{active} \Rightarrow \text{Invisible}$

## 5.7. Visibility Check

This section describes how PostgreSQL performs a visibility check, which is the process of selecting heap tuples of the appropriate versions in a given transaction. This section also describes how PostgreSQL prevents the anomalies defined in the ANSI SQL-92 Standard: Dirty Reads, Repeatable Reads and Phantom Reads.

### 5.7.1. Visibility Check

Figure 5.10 shows a scenario to describe the visibility check.

**Fig. 5.10. Scenario to describe visibility check.**

	txid = 200					txid = 201
	Tuple_1					
	t_xmin	t_xmax	t_cid	t_ctid	user data	
T1	BEGIN;					txid = 201
T2						BEGIN;
T3	SELECT * FROM tbl; snapshot="200:200;"					SELECT * FROM tbl; snapshot="200:200;"
T4	UPDATE tbl SET data = 'Hyde';	txid = 200				
		Tuple_1				
		t_xmin	t_xmax	t_cid	t_ctid	user data
		199	0	(0,1)	'Jekyll'	
		Tuple_2				
		200	0	0	(0,2)	'Hyde'
T5	SELECT * FROM tbl; snapshot="200:200;"					SELECT * FROM tbl; snapshot="200:200;"
T6	COMMIT;					
T7						SELECT * FROM tbl; snapshot = { "201:201:" if READ COMMITTED "200:200:" if REPEATABLE READ }

In the scenario shown in Fig. 5.10, SQL commands are executed in the following time sequence.

**T1:** Start transaction (txid 200)

**T2:** Start transaction (txid 201)

**T3:** Execute SELECT commands of txid 200 and 201

**T4:** Execute UPDATE command of txid 200

**T5:** Execute SELECT commands of txid 200 and 201

**T6:** Commit txid 200

**T7:** Execute SELECT command of txid 201

To simplify the description, assume that there are only two transactions, i.e. txid 200 and 201. The isolation level of txid 200 is READ COMMITTED, and the isolation level of txid 201 is either READ COMMITTED or REPEATABLE READ.

We explore how SELECT commands perform a visibility check for each tuple.

**SELECT commands of T3:**

At T3, there is only Tuple\_1 in the table *tbl* and it is *visible* by **Rule 6**. Therefore, SELECT commands in both transactions return 'Jekyll'.

- Rule6(Tuple\_1)  $\Rightarrow$  Status(t\_xmin:199) = COMMITTED  $\wedge$  t\_xmax = INVALID  $\Rightarrow$  Visible

```
testdb=# -- txid 200
testdb=# SELECT * FROM tbl;
 name
```

**Jekyll**  
(1 row)

```
testdb=# -- txid 201
testdb=# SELECT * FROM tbl;
 name
```

**Jekyll**  
(1 row)

#### SELECT commands of T5:

First, we explore the SELECT command executed by txid 200. Tuple\_1 is invisible by **Rule 7** and Tuple\_2 is visible by **Rule 2**. Therefore, this SELECT command returns 'Hyde'.

- Rule7(Tuple\_1): Status(t\_xmin:199) = COMMITTED  $\wedge$  Status(t\_xmax:200) = IN\_PROGRESS  $\wedge$  t\_xmax:200 = current\_txid:200  $\Rightarrow$  Invisible
- Rule2(Tuple\_2): Status(t\_xmin:200) = IN\_PROGRESS  $\wedge$  t\_xmin:200 = current\_txid:200  $\wedge$  t\_xmax = INVALID  $\Rightarrow$  Visible

```
testdb=# -- txid 200
testdb=# SELECT * FROM tbl;
 name
```

**Hyde**  
(1 row)

On the other hand, in the SELECT command executed by txid 201, Tuple\_1 is visible by **Rule 8** and Tuple\_2 is invisible by **Rule 4**. Therefore, this SELECT command returns 'Jekyll'.

- Rule8(Tuple\_1): Status(t\_xmin:199) = COMMITTED  $\wedge$  Status(t\_xmax:200) = IN\_PROGRESS  $\wedge$  t\_xmax:200  $\neq$  current\_txid:201  $\Rightarrow$  Visible
- Rule4(Tuple\_2): Status(t\_xmin:200) = IN\_PROGRESS  $\wedge$  t\_xmin:200  $\neq$  current\_txid:201  $\Rightarrow$  Invisible

```
testdb=# -- txid 201
testdb=# SELECT * FROM tbl;
 name
```

**Jekyll**  
(1 row)

If the updated tuples are visible from other transactions before they are committed, this is known as **Dirty Reads**, also known as **wr-conflicts**. However, as shown above, Dirty Reads do not occur in any isolation levels in PostgreSQL.

#### SELECT command of T7:

In the following, the behaviors of SELECT commands of T7 in both isolation levels are described.

When txid 201 is in the READ COMMITTED level, txid 200 is treated as COMMITTED because the transaction snapshot is '201:201:'. Therefore, Tuple\_1 is *invisible* by **Rule 10** and Tuple\_2 is *visible* by **Rule 6**. The SELECT command returns 'Hyde'.

- Rule10(Tuple\_1): Status(t\_xmin:199) = COMMITTED  $\wedge$  Status(t\_xmax:200) = COMMITTED  $\wedge$  Snapshot(t\_xmax:200)  $\neq$  active  $\Rightarrow$  Invisible

- Rule6(Tuple\_2):  $\text{Status}(\text{t\_xmin}:200) = \text{COMMITTED} \wedge \text{t\_xmax} = \text{INVALID} \Rightarrow \text{Visible}$

```
testdb=# -- txid 201 (READ COMMITTED)
testdb=# SELECT * FROM tbl;
name
-----
Hyde
(1 row)
```

Note that the results of the SELECT commands, which are executed before and after txid 200 is committed, differ. This is generally known as **Non-Repeatable Reads**.

In contrast, when txid 201 is in the REPEATABLE READ level, txid 200 must be treated as IN\_PROGRESS because the transaction snapshot is '200:200:'. Therefore, Tuple\_1 is *visible* by **Rule 9** and Tuple\_2 is *invisible* by **Rule 5**. The SELECT command returns 'Jekyll'. Note that Non-Repeatable Reads do not occur in the REPEATABLE READ (and SERIALIZABLE) level.

- Rule9(Tuple\_1):  $\text{Status}(\text{t\_xmin}:199) = \text{COMMITTED} \wedge \text{Status}(\text{t\_xmax}:200) = \text{COMMITTED} \wedge \text{Snapshot}(\text{t\_xmax}:200) = \text{active} \Rightarrow \text{Visible}$
- Rule5(Tuple\_2):  $\text{Status}(\text{t\_xmin}:200) = \text{COMMITTED} \wedge \text{Snapshot}(\text{t\_xmin}:200) = \text{active} \Rightarrow \text{Invisible}$

```
testdb=# -- txid 201 (REPEATABLE READ)
testdb=# SELECT * FROM tbl;
name
-----
Jekyll
(1 row)
```

## 1 Hint Bits

To obtain the status of a transaction, PostgreSQL internally provides three functions: TransactionIdIsInProgress, TransactionIdDidCommit, and TransactionIdDidAbort. These functions are implemented to reduce frequent access to the clog, such as caches. However, bottlenecks will occur if they are executed whenever each tuple is checked.

To deal with this issue, PostgreSQL uses *hint bits*, which are shown below:

```
#define HEAP_XMIN_COMMITTED      0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID        0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITTED      0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID        0x0800 /* t_xmax invalid/aborted */
```

When reading or writing a tuple, PostgreSQL sets hint bits to the t\_infomask of the tuple if possible. For example, assume that PostgreSQL checks the status of the t\_xmin of a tuple and obtains the status COMMITTED. In this case, PostgreSQL sets a hint bit HEAP\_XMIN\_COMMITTED to the t\_infomask of the tuple. If hint bits are already set, TransactionIdDidCommit and TransactionIdDidAbort are no longer needed. Therefore, PostgreSQL can efficiently check the statuses of both t\_xmin and t\_xmax of each tuple.

## 5.7.2. Phantom Reads in PostgreSQL's REPEATABLE READ Level

REPEATABLE READ as defined in the ANSI SQL-92 standard allows **Phantom Reads**. However, PostgreSQL's implementation does not allow them. In principle, SI does not allow Phantom Reads.

Assume that two transactions, i.e. Tx\_A and Tx\_B, are running concurrently. Their isolation levels are READ COMMITTED and REPEATABLE READ, and their txids are 100 and 101, respectively. First, Tx\_A inserts a tuple. Then, it is committed. The t\_xmin of the inserted tuple is 100. Next,

Tx\_B executes a SELECT command; however, the tuple inserted by Tx\_A is *invisible* by **Rule 5**. Thus, Phantom Reads do not occur.

- Rule5(new tuple): Status(t\_xmin:100) = COMMITTED  $\wedge$  Snapshot(t\_xmin:100) = active  $\Rightarrow$  Invisible

```
testdb=# -- Tx_A: txid 100
testdb=# START TRANSACTION
testdb=# ISOLATION LEVEL READ COMMITTED;
START TRANSACTION
```

```
testdb=# INSERT tbl(id, data)
          VALUES (1,'phantom');
INSERT 1

testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B: txid 101
testdb=# START TRANSACTION
testdb=# ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
testdb=# SELECT txid_current();
txid_current
```

```
101
(1 row)
```

```
testdb=# SELECT * FROM tbl WHERE id=1;
 id | data
----+
 (0 rows)
```

## 5.8. Preventing Lost Updates

A **Lost Update**, also known as a **ww-conflict**, is an anomaly that occurs when concurrent transactions update the same rows, and it must be prevented in both the REPEATABLE READ and SERIALIZABLE levels. (Note that the READ COMMITTED level does not need to prevent Lost Updates.) This section describes how PostgreSQL prevents Lost Updates and shows examples.

### 5.8.1. Behavior of Concurrent UPDATE Commands

When an UPDATE command is executed, the function ExecUpdate is internally invoked. The pseudocode of the ExecUpdate is shown below:

#### </> Pseudocode: ExecUpdate

```
(1) FOR each row that will be updated by this UPDATE command
(2)   WHILE true
        /* The First Block */
        IF the target row is being updated THEN
            WAIT for the termination of the transaction that updated the target row
        (3)       IF (the status of the terminated transaction is COMMITTED)
                  AND (the isolation level of this transaction is REPEATABLE READ or SERIALIZABLE) TH
        EN
```

```

(6)          ABORT this transaction /* First-Updater-Win */
ELSE
(7)          GOTO step (2)
END IF

/* The Second Block */
(8) ELSE IF the target row has been updated by another concurrent transaction THEN
(9)   IF (the isolation level of this transaction is READ COMMITTED THEN
(10)    UPDATE the target row
ELSE
(11)   ABORT this transaction /* First-Updater-Win */
END IF

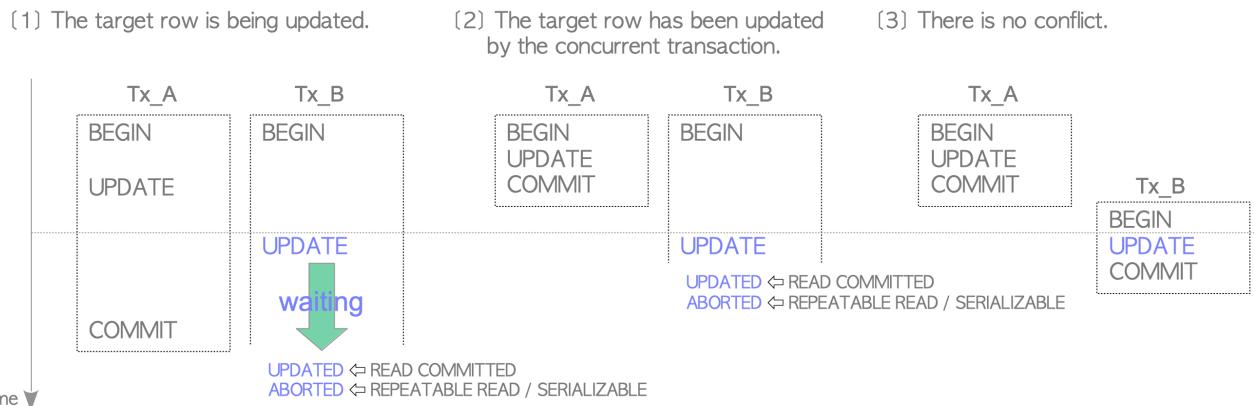
/* The Third Block */
ELSE /* The target row is not yet modified or has been updated by a terminated transaction.
*/
(12) UPDATE the target row
END IF
END WHILE
END FOR

```

- (1) Get each row that will be updated by this UPDATE command.
- (2) Repeat the following process until the target row has been updated (or this transaction is aborted).
  - (3) If the target row is being updated, go to step (3); otherwise, go to step (8).
  - (4) Wait for the termination of the transaction that updated the target row because PostgreSQL uses *first-updater-win* scheme in SI.
  - (5) If the status of the transaction that updated the target row is COMMITTED and the isolation level of this transaction is REPEATABLE READ (or SERIALIZABLE), go to step (6); otherwise, go to step (7).
  - (6) Abort this transaction to prevent Lost Updates.
  - (7) Go to step (2) and attempt to update the target row in the next round.
  - (8) If the target row has been updated by another concurrent transaction, go to step (9); otherwise, go to step (12).
  - (9) If the isolation level of this transaction is READ COMMITTED, go to step (10); otherwise, go to step (11).
  - (10) UPDATE the target row, and go to step (1).
  - (11) Abort this transaction to prevent Lost Updates.
  - (12) UPDATE the target row, and go to step (1) because the target row is not yet modified or has been updated by a terminated transaction, i.e. there is ww-conflict.

This function performs update operations for each of the target rows. It has a while loop to update each row, and the inside of the while loop branches to three blocks according to the conditions shown in Fig. 5.11.

**Fig. 5.11. Three internal blocks in ExecUpdate.**



[1] The target row is being updated (Fig. 5.11[1])

'Being updated' means that the row is being updated by another concurrent transaction and its transaction has not terminated. In this case, the current transaction must wait for termination of the transaction that updated the target row because PostgreSQL's SI uses the **first-updater-win** scheme. For example, assume that transactions Tx\_A and Tx\_B run concurrently, and Tx\_B attempts to update a row; however, Tx\_A has updated it and is still in progress. In this case, Tx\_B waits for the termination of Tx\_A.

After the transaction that updated the target row commits, the update operation of the current transaction proceeds. If the current transaction is in the READ COMMITTED level, the target row will be updated; otherwise (REPEATABLE READ or SERIALIZABLE), the current transaction is aborted immediately to prevent lost updates.

- [2] The target row has been updated by the concurrent transaction (Fig. 5.11[2])

The current transaction attempts to update the target tuple; however, the other concurrent transaction has updated the target row and has already been committed. In this case, if the current transaction is in the READ COMMITTED level, the target row will be updated; otherwise, the current transaction is aborted immediately to prevent lost updates.

- [3] There is no conflict (Fig. 5.11[3])

When there is no conflict, the current transaction can update the target row.

### ❶ first-updater-win / first-committer-win

As mentioned in this section, PostgreSQL's concurrency control based on SI uses the *first-updater-win* scheme to avoid lost update anomalies. In contrast, as explained in the next section, PostgreSQL's SSI uses the *first-committer-win* scheme to avoid serialization anomalies.

## 5.8.2. Examples

Three examples are shown in the following. The first and second examples show behaviours when the target row is being updated, and the third example shows the behaviour when the target row has been updated.

### Example 1:

Transactions Tx\_A and Tx\_B update the same row in the same table, and their isolation level is READ COMMITTED.

```
testdb=# -- Tx_A
testdb=# START TRANSACTION
testdb=#   ISOLATION LEVEL READ COMMITTED;
START TRANSACTION
```

```
testdb=# UPDATE tbl SET name = 'Hyde';
UPDATE 1
```

```
testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B
testdb=# START TRANSACTION
testdb=#   ISOLATION LEVEL READ COMMITTED;
START TRANSACTION
```

```

testdb=# UPDATE tbl SET name = 'Utterson';
↓
↓ this transaction is being blocked
↓
UPDATE 1

```

Tx\_B is executed as follows.

- 1) After executing the UPDATE command, Tx\_B should wait for the termination of Tx\_A, because the target tuple is being updated by Tx\_A (Step (4) in ExecUpdate).
- 2) After Tx\_A is committed, Tx\_B attempts to update the target row (Step (7) in ExecUpdate).
- 3) In the second round of ExecUpdate, the target row is updated again by Tx\_B (Steps (2),(8),(9), (10) in ExecUpdate).

#### **Example 2:**

Tx\_A and Tx\_B update the same row in the same table, and their isolation levels are READ COMMITTED and REPEATABLE READ, respectively.

```

testdb=# -- Tx_A
testdb=# START TRANSACTION
testdb-#   ISOLATION LEVEL READ COMMITTED;
START TRANSACTION

```

```

testdb=# UPDATE tbl SET name = 'Hyde';
UPDATE 1

```

```

testdb=# COMMIT;
COMMIT

```

```

testdb=# -- Tx_B
testdb=# START TRANSACTION
testdb-#   ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION

```

```

testdb=# UPDATE tbl SET name = 'Utterson';
↓
↓ this transaction is being blocked
↓
ERROR:couldn't serialize access due to concurrent update

```

The behaviour of Tx\_B is described as follows.

- 1) After executing the UPDATE command, Tx\_B should wait for the termination of Tx\_A (Step (4) in ExecUpdate).
- 2) After Tx\_A is committed, Tx\_B is aborted to resolve conflict because the target row has been updated and the isolation level of this transaction is REPEATABLE READ (Steps (5) and (6) in ExecUpdate).

#### **Example 3:**

Tx\_B (REPEATABLE READ) attempts to update the target row that has been updated by the committed Tx\_A. In this case, Tx\_B is aborted (Steps (2),(8),(9), and (11) in ExecUpdate).

```

testdb=# -- Tx_A
testdb=# START TRANSACTION

```

```
testdb=# ISOLATION LEVEL READ COMMITTED;
START TRANSACTION
```

```
testdb=# UPDATE tbl SET name = 'Hyde';
UPDATE 1
```

```
testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B
testdb=# START TRANSACTION
testdb=# ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
testdb=# SELECT * FROM tbl;
name
```

### Jekyll

(1 row)

```
testdb=# UPDATE tbl SET name = 'Utterson';
ERROR:couldn't serialize access due to concurrent update
```

## 5.9. Serializable Snapshot Isolation

Serializable Snapshot Isolation (SSI) has been embedded in SI since version 9.1 to realize a true SERIALIZABLE isolation level. Since the explanation of SSI is not simple, only an outline is explained. For details, see [2].

In the following, the technical terms shown in below are used without definitions. If you are unfamiliar with these terms, see [1, 3].

- *precedence graph* (also known as *dependency graph* and *serialization graph*)
- *serialization anomalies* (e.g. *Write-Skew*)

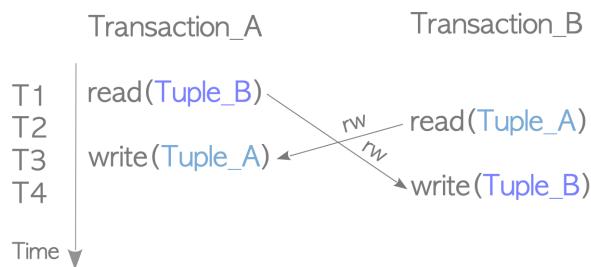
### 5.9.1. Basic Strategy for SSI Implementation

If a cycle is present in the precedence graph, there will be a serialization anomaly. This can be explained using the simplest anomaly, write-skew.

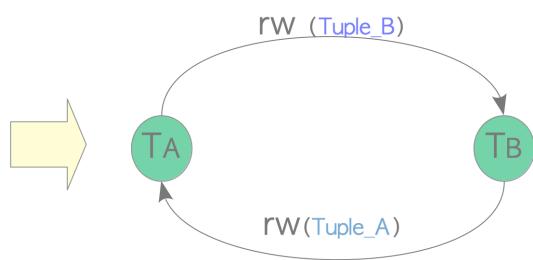
Figure 5.12(1) shows a schedule. Here, Transaction\_A reads Tuple\_B and Transaction\_B reads Tuple\_A. Then, Transaction\_A writes Tuple\_A and Transaction\_B writes Tuple\_B. In this case, there are two rw-conflicts, and they make a cycle in the precedence graph of this schedule, as shown in Fig. 5.12(2). Thus, this schedule has a serialization anomaly, Write-Skew.

**Fig. 5.12. Write-Skew schedule and its precedence graph.**

(1) a write-skew schedule



(2) the precedence graph of schedule (1)



Conceptually, there are three types of conflicts: wr-conflicts (Dirty Reads), ww-conflicts (Lost Updates), and rw-conflicts. However, wr- and ww-conflicts do not need to be considered because PostgreSQL prevents such conflicts, as shown in the previous sections. Thus, SSI implementation in PostgreSQL only needs to consider rw-conflicts.

PostgreSQL takes the following strategy for the SSI implementation:

1. Record all objects (tuples, pages, relations) accessed by transactions as SIREAD locks.
2. Detect rw-conflicts using SIREAD locks whenever any heap or index tuple is written.
3. Abort the transaction if a serialization anomaly is detected by checking detected rw-conflicts.

## 5.9.2. Implementing SSI in PostgreSQL

To realize the strategy described above, PostgreSQL has implemented many functions and data structures. However, here we uses only two data structures: **SIREAD locks** and **rw-conflicts**, to describe the SSI mechanism. They are stored in shared memory.



For simplicity, some important data structures, such as SERIALIZABLEXACT, are omitted in this document. Thus, the explanations of the functions, i.e. CheckForSerializableConflictOut, CheckForSerializableConflictIn, and PreCommit\_CheckForSerializationFailure, are also extremely simplified. For example, we indicate which functions detect conflicts; however, how the conflicts are detected is not explained in detail. If you want to know the details, refer to the source code: `src/backend/storage/lmgr/predicate.c`.

### SIREAD locks:

An SIREAD lock, internally called a predicate lock, is a pair of an object and (virtual) txids that store information about who has accessed which object. Note that the description of virtual txid is omitted. The term txid is used rather than virtual txid to simplify the following explanation.

SIREAD locks are created by the CheckForSerializableConflictOut function whenever a DML command is executed in SERIALIZABLE mode. For example, if txid 100 reads Tuple\_1 of the given table, an SIREAD lock `{Tuple_1, {100}}` is created. If another transaction, e.g. txid 101, reads Tuple\_1, the SIREAD lock is updated to `{Tuple_1, {100,101}}`. Note that a SIREAD lock is also created when an index page is read because an index page is only read without reading the table page when the Index-Only Scans feature, which is described in Section 7.2, is applied.

SIREAD lock has three levels: tuple, page, and relation. If the SIREAD locks of all tuples within a single page are created, they are aggregated into a single SIREAD lock for that page, and all SIREAD locks of the associated tuples are released (removed), to reduce memory space. The same is true for all pages that are read.

When using sequential scan, a relation level SIREAD lock is created from the beginning regardless of the presence of indexes and/or WHERE clauses. Note that, in certain situations, this implementation can cause false-positive detections of serialization anomalies. The details are described in Section 5.9.4.

### rw-conflicts:

A rw-conflict is a triplet of an SIREAD lock and two txids that reads and writes the SIREAD lock.

The CheckForSerializableConflictIn function is invoked whenever either an INSERT, UPDATE, or DELETE command is executed in SERIALIZABLE mode, and it creates rw-conflicts when detecting conflicts by checking SIREAD locks.

For example, assume that txid 100 reads Tuple\_1 and then txid 101 updates Tuple\_1. In this case, the CheckForSerializableConflictIn function, invoked by the UPDATE command in txid 101, detects a rw-conflict with Tuple\_1 between txid 100 and 101 and then creates a rw-conflict {r=100, w=101, {Tuple\_1}}.

Both the CheckForSerializableConflictOut and CheckForSerializableConflictIn functions, as well as the PreCommit\_CheckForSerializationFailure function, which is invoked when the COMMIT command is executed in SERIALIZABLE mode, check serialization anomalies using the created rw-conflicts. If they detect anomalies, only the first-committed transaction is committed and the other transactions are aborted (by the **first-committer-win** scheme).

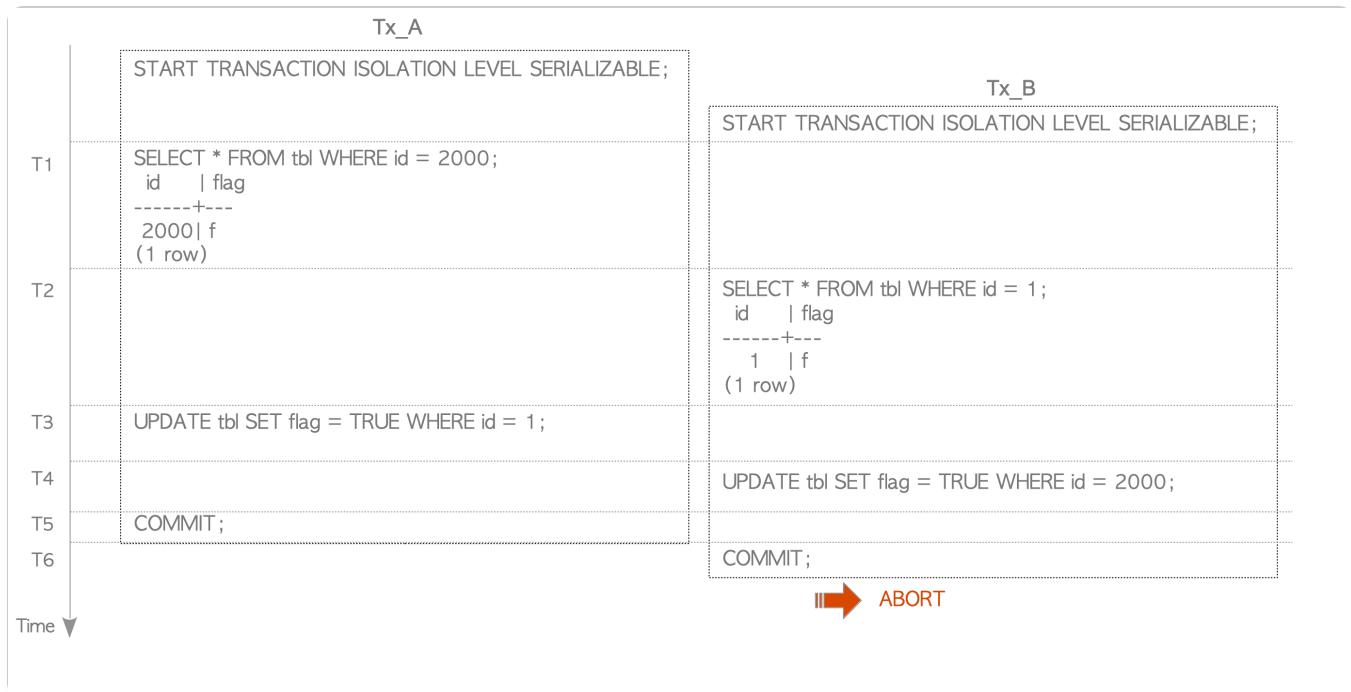
### 5.9.3. How SSI Performs

Here, we describe how SSI resolves Write-Skew anomalies. We use a simple table *tbl* shown below:

```
testdb=# CREATE TABLE tbl (id INT primary key, flag bool DEFAULT false);
testdb=# INSERT INTO tbl (id) SELECT generate_series(1,2000);
testdb=# ANALYZE tbl;
```

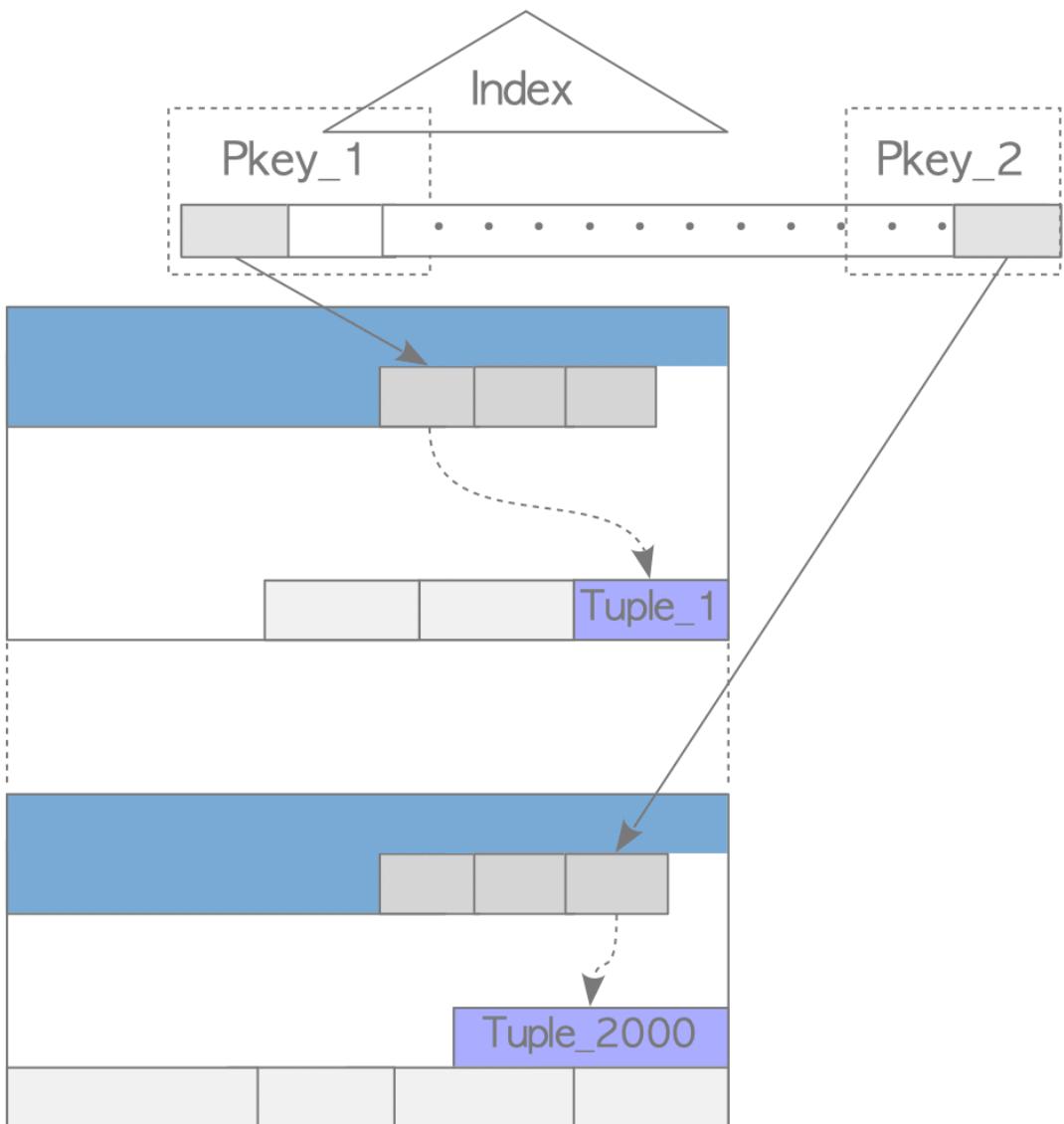
Transactions Tx\_A and Tx\_B execute the following commands (Fig. 5.13).

**Fig. 5.13. Write-Skew scenario.**



Assume that all commands use index scan. Therefore, when the commands are executed, they read both heap tuples and index pages, each of which contains the index tuple that points to the corresponding heap tuple. See Fig. 5.14.

**Fig. 5.14. Relationship between the index and table in the scenario shown in Fig. 5.13.**



**T1:** Tx\_A executes a SELECT command. This command reads a heap tuple (Tuple\_2000) and one page of the primary key (Pkey\_2).

**T2:** Tx\_B executes a SELECT command. This command reads a heap tuple (Tuple\_1) and one page of the primary key (Pkey\_1).

**T3:** Tx\_A executes an UPDATE command to update Tuple\_1.

**T4:** Tx\_B executes an UPDATE command to update Tuple\_2000.

**T5:** Tx\_A commits.

**T6:** Tx\_B commits; however, it is aborted due to a Write-Skew anomaly.

Figure 5.15 shows how PostgreSQL detects and resolves the Write-Skew anomaly described in the above scenario.

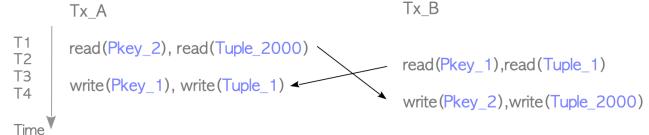
**Fig. 5.15. SIREAD locks and rw-conflicts, and schedule of the scenario shown in Fig. 5.13.**

(1) SIREAD Locks and rw-conflicts shown in Figure 5.13

	SIREAD Locks	rw-conflicts
T1	L1: {Pkey_2, {Tx_A}} L2: {Tuple_2000, {Tx_A}}	
T2	L1: {Pkey_2, {Tx_A}} L2: {Tuple_2000, {Tx_A}} L3: {Pkey_1, {Tx_B}} L4: {Tuple_1, {Tx_B}}	C1: {r=Tx_B, w=Tx_A, {Pkey_1, Tuple_1}}
T3		
T4		C1: {r=Tx_B, w=Tx_A, {Pkey_1, Tuple_1}} C2: {r=Tx_A, w=Tx_B, {Pkey_2, Tuple_2000}}
T5		
T6		

Time ↓

(2) Schedule shown in Figure 5.13

**T1:**

When executing the SELECT command of Tx\_A, CheckForSerializableConflictOut creates SIREAD locks. In this scenario, the function creates two SIREAD locks: L1 and L2. L1 and L2 are associated with Pkey\_2 and Tuple\_2000, respectively.

**T2:**

When executing the SELECT command of Tx\_B, CheckForSerializableConflictOut creates two SIREAD locks: L3 and L4.

L3 and L4 are associated with Pkey\_1 and Tuple\_1, respectively.

**T3:**

When executing the UPDATE command of Tx\_A, both CheckForSerializableConflictOut and CheckTargetForConflictsIN are invoked before and after ExecUpdate.

In this scenario, CheckForSerializableConflictOut does nothing.

CheckForSerializableConflictIn creates rw-conflict C1, which is the conflict of both Pkey\_1 and Tuple\_1 between Tx\_B and Tx\_A, because both Pkey\_1 and Tuple\_1 were read by Tx\_B and written by Tx\_A.

**T4:**

When executing the UPDATE command of Tx\_B, CheckForSerializableConflictIn creates rw-conflict C2, which is the conflict of both Pkey\_2 and Tuple\_2000 between Tx\_A and Tx\_B.

In this scenario, C1 and C2 create a cycle in the precedence graph; thus, Tx\_A and Tx\_B are in a non-serializable state. However, both transactions Tx\_A and Tx\_B have not been committed, therefore CheckForSerializableConflictIn does not abort Tx\_B. Note that this occurs because PostgreSQL's SSI implementation is based on the *first-committer-win* scheme.

**T5:**

When Tx\_A attempts to commit, PreCommit\_CheckForSerializationFailure is invoked. This function can detect serialization anomalies and can execute a commit action if possible. In this scenario, Tx\_A is committed because Tx\_B is still in progress.

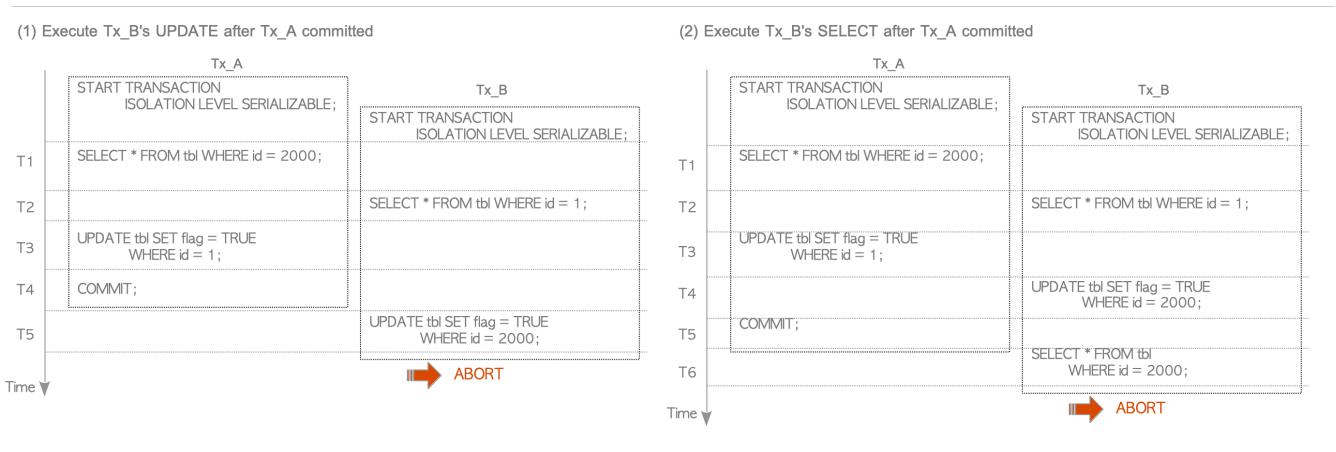
**T6:**

When Tx\_B attempts to commit, PreCommit\_CheckForSerializationFailure detects a serialization anomaly and Tx\_A has already been committed; thus, Tx\_B is aborted.

In addition, if the UPDATE command is executed by Tx\_B after Tx\_A has been committed (at **T5**), Tx\_B is immediately aborted because CheckForSerializableConflictIn invoked by Tx\_B's UPDATE command detects a serialization anomaly (Fig. 5.16(1)).

If the SELECT command is executed instead of COMMIT at **T6**, Tx\_B is immediately aborted because CheckForSerializableConflictOut invoked by Tx\_B's SELECT command detects a serialization anomaly (Fig. 5.16(2)).

**Fig. 5.16. Other Write-Skew scenarios.**



This Wiki explains several more complex anomalies.

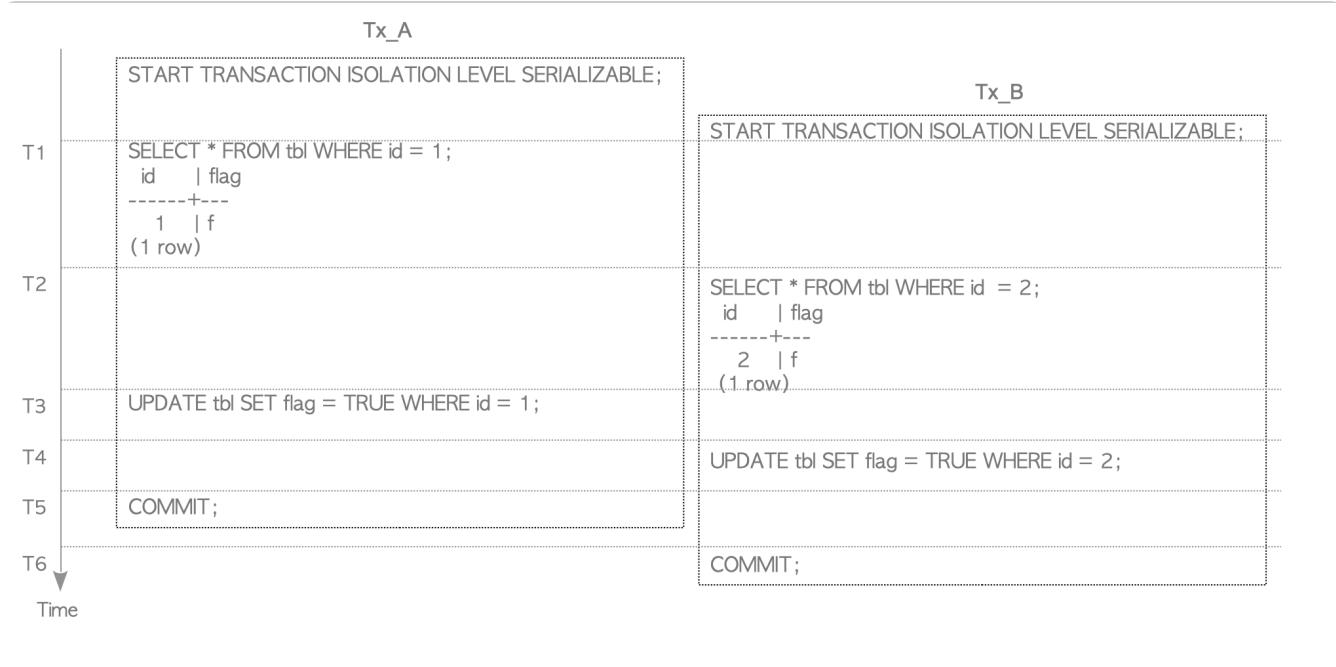
## 5.9.4. False-Positive Serialization Anomalies

In SERIALIZABLE mode, the serializability of concurrent transactions is always fully guaranteed because false-negative serialization anomalies are never detected. However, under some circumstances, false-positive anomalies can be detected; therefore, users should keep this in mind when using SERIALIZABLE mode. In the following, the situations in which PostgreSQL detects false-positive anomalies are described.

### 5.9.4.1. False-Positive Scenario 1.

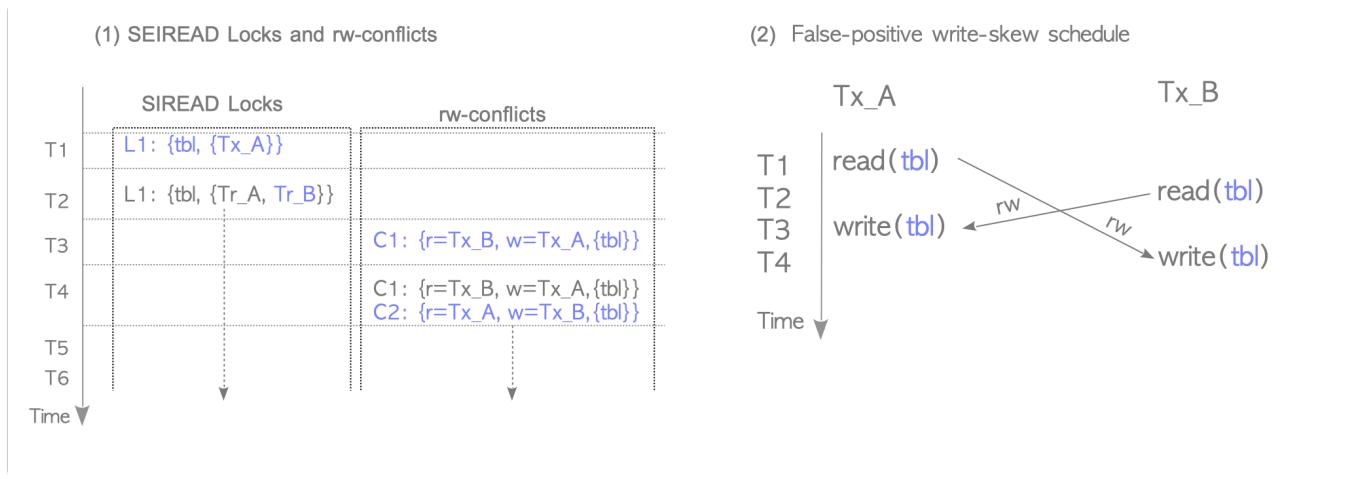
Figure 5.17 shows a scenario where a false-positive serialization anomaly occurs.

**Fig. 5.17. Scenario where false-positive serialization anomaly occurs.**



When using sequential scan, as mentioned in the explanation of SIREAD locks, PostgreSQL creates a relation level SIREAD lock. Figure 5.18(1) shows SIREAD locks and rw-conflicts when PostgreSQL uses sequential scan. In this case, rw-conflicts C1 and C2, which are associated with the tbl's SIREAD lock, are created, and they create a cycle in the precedence graph. Thus, a false-positive Write-Skew anomaly is detected (and either Tx\_A or Tx\_B will be aborted even though there is no conflict).

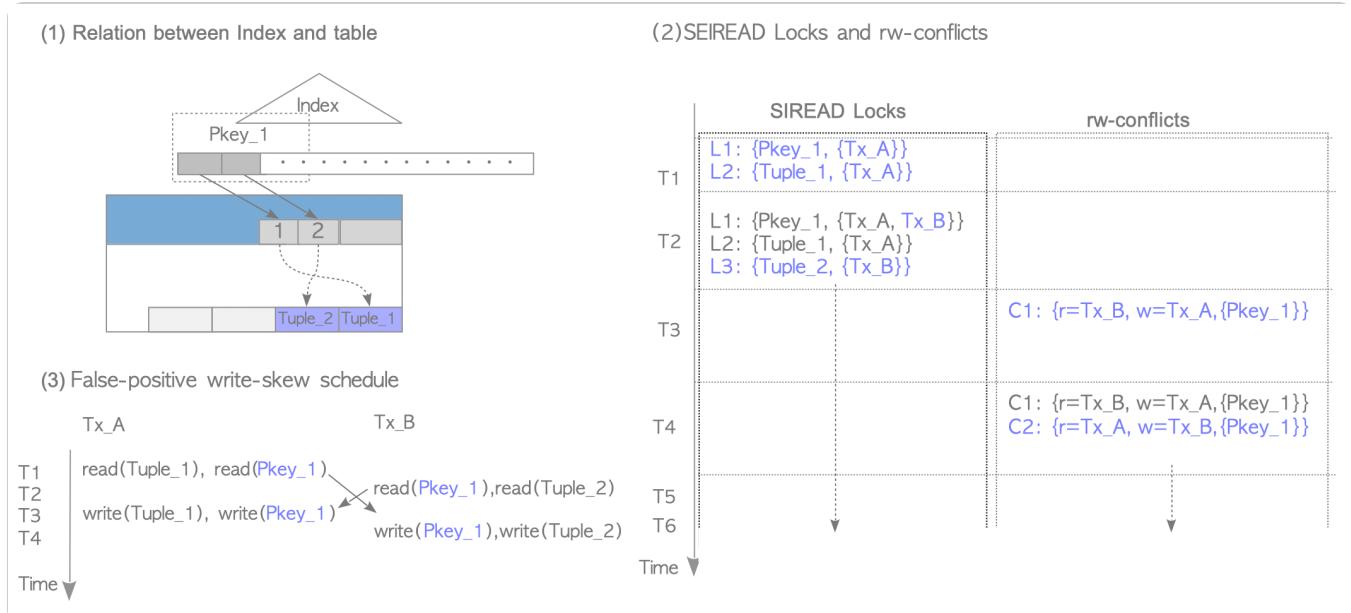
**Fig. 5.18. False-positive anomaly (1) – Using sequential scan.**



### 5.9.4.2. False-Positive Scenario 2.

Even when using index scan, if both transactions Tx\_A and Tx\_B get the same index SIREAD lock, PostgreSQL detects a false-positive anomaly. Figure 5.19 shows this situation.

**Fig. 5.19. False-positive anomaly (2) – Index scan using the same index page.**



Assume that the index page Pkey\_1 contains two index items, one of which points to Tuple\_1 and the other points to Tuple\_2.

When Tx\_A and Tx\_B execute respective SELECT and UPDATE commands, Pkey\_1 is read and written by both Tx\_A and Tx\_B. In this case, rw-conflicts C1 and C2, both of which are associated with Pkey\_1, create a cycle in the precedence graph; thus, a false-positive Write-Skew anomaly is detected.

(If Tx\_A and Tx\_B get the SIREAD locks of different index pages, a false-positive is not detected and both transactions can be committed.)

## 5.10. Required Maintenance Processes

PostgreSQL's concurrency control mechanism requires the following maintenance processes:

1. Remove dead tuples and index tuples that point to corresponding dead tuples
2. Remove unnecessary parts of the clog
3. Freeze old txids
4. Update FSM, VM, and the statistics

The need for the first and second processes have been explained in Sections 5.3.2 and 5.4.3, respectively. The third process is related to the transaction id wraparound problem, which is briefly described in the following subsection.

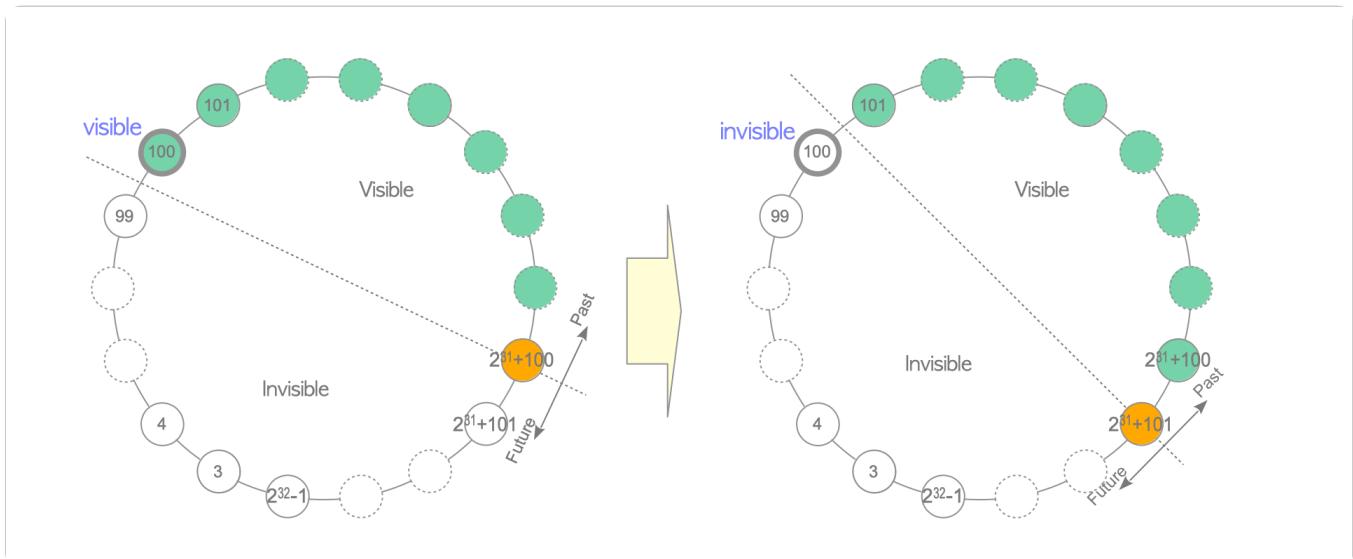
In PostgreSQL, **VACUUM** processing is responsible for these processes, and it is described in Chapter 6.

### 5.10.1. FREEZE Processing

Here, I describe the txid wraparound problem.

Assume that tuple Tuple\_1 is inserted with a txid of 100, i.e. the `t_xmin` of Tuple\_1 is 100. The server has been running for a very long period and Tuple\_1 has not been modified. The current txid is 2.1 billion + 100 and a `SELECT` command is executed. At this time, Tuple\_1 is *visible* because txid 100 is *in the past*. Then, the same `SELECT` command is executed; thus, the current txid is 2.1 billion + 101. However, Tuple\_1 is *no longer visible* because txid 100 is *in the future* (Fig. 5.20). This is the so called *transaction wraparound problem* in PostgreSQL.

**Fig. 5.20. Wraparound problem.**



To deal with this problem, PostgreSQL introduced a concept called *frozen txid*, and implemented a process called *FREEZE*.

In PostgreSQL, a frozen txid, which is a special reserved txid 2, is defined such that it is always older than all other txids. In other words, the frozen txid is always inactive and visible.

The freeze process is invoked by the vacuum process. The freeze process scans all table files and rewrites the t\_xmin of tuples to the frozen txid(2) if the t\_xmin value is older than the current txid minus the vacuum\_freeze\_min\_age (the default is 50 million). This is explained in more detail in Chapter 6.

For example, as can be seen in Fig. 5.21 a), the current txid is 50 million and the freeze process is invoked by the VACUUM command. In this case, the t\_xmin of both Tuple\_1 and Tuple\_2 are rewritten to 2.

In versions 9.4 or later, the XMIN\_FROZEN bit is set to the t\_infomask field of tuples rather than rewriting the t\_xmin of tuples to the frozen txid (Fig. 5.21 b).

**Fig. 5.21. Freeze process.**

a) Version 9.3 or earlier

	t_xmin	t_xmax	t_infomask	user data
Tuple 1	99			'A'
Tuple 2	100			'B'
Tuple 3	200000			'C'
Tuple 4	1.5 million			'D'
Tuple 5	2.0 million			'E'



	t_xmin	t_xmax	t_infomask	user data
	2			'A'
	2			'B'
	200000			'C'
	1.5 million			'D'
	2.0 million			'E'

b) Version 9.4 or later

	t_xmin	t_xmax	t_infomask	user data
Tuple 1	99			'A'
Tuple 2	100			'B'
Tuple 3	200000			'C'
Tuple 4	1.5 million			'D'
Tuple 5	2.0 million			'E'



	t_xmin	t_xmax	t_infomask	user data
	99		XMIN_FROZEN	'A'
	100		XMIN_FROZEN	'B'
	200000			'C'
	1.5 million			'D'
	2.0 million			'E'

## References

- [1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, "Database System Concepts", McGraw-Hill Education, ISBN-13: 978-0073523323
- [2] Dan R. K. Ports, and Kevin Grittner, "Serializable Snapshot Isolation in PostgreSQL", VDBL 2012
- [3] Thomas M. Connolly, and Carolyn E. Begg, "Database Systems", Pearson, ISBN-13: 978-0321523068

# Chapter 6

## Vacuum Processing

---

**V**acuum processing is a maintenance process that facilitates the persistent operation of PostgreSQL. Its two main tasks are *removing dead tuples* and the *freezing transaction ids*, both of which are briefly mentioned in Section 5.10.

To remove dead tuples, vacuum processing provides two modes, namely **Concurrent VACUUM** and **Full VACUUM**. Concurrent VACUUM, often simply called VACUUM, removes dead tuples for each page of the table file, and other transactions can read the table while this process is running. In contrast, Full VACUUM removes dead tuples and defragments live tuples in the whole file, and other transactions cannot access tables while Full VACUUM is running.

Despite the fact that vacuum processing is essential for PostgreSQL, improving its functionality has been slow compared to other functions. For example, until version 8.0, this process had to be executed manually (with the psql utility or using the cron daemon). It was automated in 2005 when the **autovacuum** daemon was implemented.

Since vacuum processing involves scanning whole tables, it is a costly process. In version 8.4 (2009), the **Visibility Map (VM)** was introduced to improve the efficiency of removing dead tuples. In version 9.6 (2016), the freeze process was improved by enhancing the VM.

Section 6.1 outlines the concurrent VACUUM process. Then, subsequent sections describe the following.

- Visibility Map
- Freeze processing
- Removing unnecessary clog files
- Autovacuum daemon
- Full VACUUM

### 6.1. Outline of Concurrent VACUUM

Vacuum processing performs the following tasks for specified tables or all tables in the database:

1. Removing dead tuples
  - Remove dead tuples and defragment live tuples for each page.
  - Remove index tuples that point to dead tuples.
2. Freezing old txids
  - Freeze old txids of tuples if necessary.
  - Update frozen txid related system catalogs (pg\_database and pg\_class).
  - Remove unnecessary parts of the clog if possible.
3. Others
  - Update the FSM and VM of processed tables.
  - Update several statistics (pg\_stat\_all\_tables, etc).

It is assumed that readers are familiar with following terms: dead tuples, freezing txid, FSM, and the clog; if you are not, refer to Chapter 5. VM is introduced in Section 6.2.

The following pseudocode describes vacuum processing.

## </> Pseudocode: Concurrent VACUUM

```
(1) FOR each table
(2)   Acquire a ShareUpdateExclusiveLock lock for the target table

  /* The first block */
(3)   Scan all pages to get all dead tuples, and freeze old tuples if necessary
(4)   Remove the index tuples that point to the respective dead tuples if exists

  /* The second block */
(5)   FOR each page of the table
(6)     Remove the dead tuples, and Reallocate the live tuples in the page
(7)     Update FSM and VM
  END FOR

  /* The third block */
(8)   Clean up indexes
(9)   Truncate the last page if possible
(10)  Update both the statistics and system catalogs of the target table
    Release the ShareUpdateExclusiveLock lock
  END FOR

  /* Post-processing */
(11) Update statistics and system catalogs
(12) Remove both unnecessary files and pages of the clog if possible
```

- (1) Get each table from the specified tables.
- (2) Acquire a ShareUpdateExclusiveLock lock for the table. This lock allows reading from other transactions.
- (3) Scan all pages to get all dead tuples, and freeze old tuples if necessary.
- (4) Remove the index tuples that point to the respective dead tuples if exists.
- (5) Do the following tasks, step (6) and (7), for each page of the table.
- (6) Remove the dead tuples and Reallocate the live tuples in the page.
- (7) Update both the respective FSM and VM of the target table.
- (8) Clean up the indexes by the index\_vacuum\_cleanup()@indexam.c function.
- (9) Truncate the last page if the last one does not have any tuple.
- (10) Update both the statistics and the system catalogs related to vacuum processing for the target table.
- (11) Update both the statistics and the system catalogs related to vacuum processing.
- (12) Remove both unnecessary files and pages of the clog if possible.

This pseudocode has two sections: a loop for each table and post-processing. The inner loop can be divided into three blocks. Each block has individual tasks.

These three blocks and the post-process are outlined in the following.

### 1 PARALLEL option

The VACUUM command has supported the PARALLEL option since version 13. If this option is set and there are multiple indexes created, the vacuuming index and cleaning index up phases are processed in parallel.

Note that this feature is only valid for the VACUUM command and is not supported by autovacuum.

## 6.1.1. First Block

This block performs freeze processing and removes index tuples that point to dead tuples.

First, PostgreSQL scans a target table to build a list of dead tuples and freeze old tuples if possible. The list is stored in the local memory called `maintenance_work_mem`. Freeze processing is described in Section 6.3.

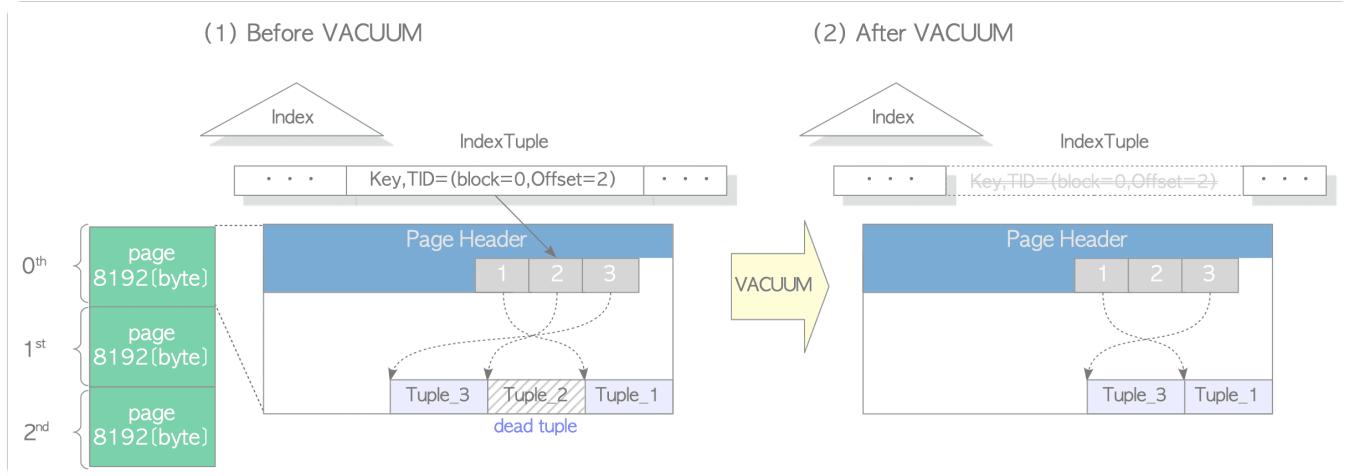
After scanning, PostgreSQL removes index tuples by referring to the dead tuple list. This process is internally called the "cleanup stage". It is a costly process, so PostgreSQL was improved in version 11. In versions 10 or earlier, the cleanup stage is always executed. In versions 11 or later, if the target index is B-tree, whether the cleanup stage is executed or not is decided by the configuration parameter `vacuum_cleanup_index_scale_factor`. See the description of this parameter in details.

If `maintenance_work_mem` is full and scanning is incomplete, PostgreSQL proceeds to the next tasks, i.e. steps (4) to (7). Then, it goes back to step (3) and proceeds remainder scanning.

## 6.1.2. Second Block

This block removes dead tuples and updates both the FSM and VM on a page-by-page basis. Figure 6.1 shows an example:

**Fig. 6.1. Removing a dead tuple.**



Assume that the table contains three pages. We focus on the 0th page (i.e., the first page). This page has three tuples. Tuple\_2 is a dead tuple (Fig. 6.1(1)). In this case, PostgreSQL removes Tuple 2 and reorders the remaining tuples to repair fragmentation. Then, it updates both the FSM and VM of this page (Fig. 6.1(2)). PostgreSQL continues this process until the last page.

Note that unnecessary line pointers are not removed. They will be reused in the future. This is because if line pointers are removed, all index tuples of the associated indexes must be updated.

## 6.1.3. Third Block

The third block performs the cleanup after the deletion of the indexes, and also updates the statistics and system catalogs related to vacuum processing for each target table.

Moreover, if the last page has no tuples, it is truncated from the table file.

## 6.1.4. Post-processing

When vacuum processing is complete, PostgreSQL updates all the statistics and system catalogs related to vacuum processing. It also removes unnecessary parts of the clog if possible (Section 6.4).

### ⓘ Ring Buffer

Vacuum processing uses a *ring buffer*, described in Section 8.5. Therefore, processed pages are not cached in the shared buffers.

## 6.2. Visibility Map

Vacuum processing is costly. Therefore, the VM was introduced in version 8.4 to reduce this cost.

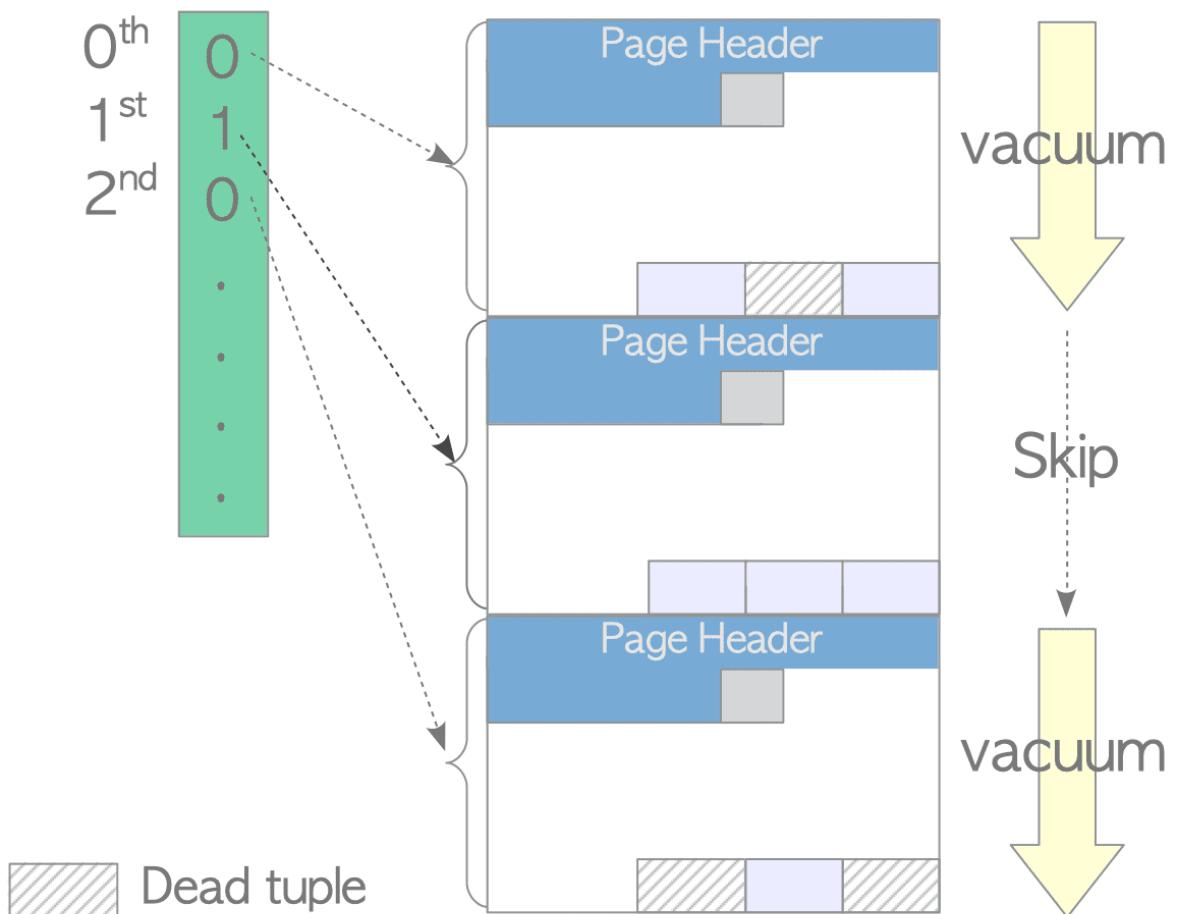
The basic concept of the VM is simple. Each table has an individual visibility map that holds the visibility of each page in the table file. The visibility of pages determines whether or not each page has dead tuples. Vacuum processing can skip a page that does not have dead tuples by using the corresponding visibility map (VM).

Figure 6.2 shows how the VM is used. Suppose that the table consists of three pages, and the 0th and 2nd pages contain dead tuples and the 1st page does not. The VM of this table holds information about which pages contain dead tuples. In this case, vacuum processing skips the 1st page by referring to the VM's information.

Each VM is composed of one or more 8 KB pages, and this file is stored with the 'vm' suffix. As an example, one table file whose relfilenode is 18751 with FSM (18751\_fsm) and VM (18751\_vm) files shown in the following.

**Fig. 6.2. How the VM is used.**

# Visibility Map



```
$ cd $PGDATA
$ ls -la base/16384/18751*
-rw----- 1 postgres postgres 8192 Apr 21 10:21 base/16384/18751
-rw----- 1 postgres postgres 24576 Apr 21 10:18 base/16384/18751_fsm
-rw----- 1 postgres postgres 8192 Apr 21 10:18 base/16384/18751_vm
```

## 6.2.1. Enhancement of VM

The VM was enhanced in version 9.6 to improve the efficiency of freeze processing. The new VM shows page visibility and information about whether tuples are frozen or not in each page (Section 6.3.3).

## 6.3. Freeze Processing

Freeze processing has two modes. For convenience, these modes are referred to as **lazy mode** and **eager mode**. It is performed in either mode depending on certain conditions.



Concurrent VACUUM is often called "lazy vacuum" internally. However, the lazy mode defined in this document is a mode of freeze processing.

Freeze processing typically runs in lazy mode, but eager mode is run when specific conditions are satisfied.

In lazy mode, freeze processing scans only pages that contain dead tuples using the respective VM of the target tables.

In contrast, eager mode scans all pages regardless of whether each page contains dead tuples or not. It also updates system catalogs related to freeze processing and removes unnecessary parts of the clog if possible.

Sections 6.3.1 and 6.3.2 describe these modes, respectively. Section 6.3.3 describes how to improve the freeze process in eager mode.

### 6.3.1. Lazy Mode

When starting freeze processing, PostgreSQL calculates the `freezeLimit_txid` and freezes tuples whose `t_xmin` is less than the `freezeLimit_txid`.

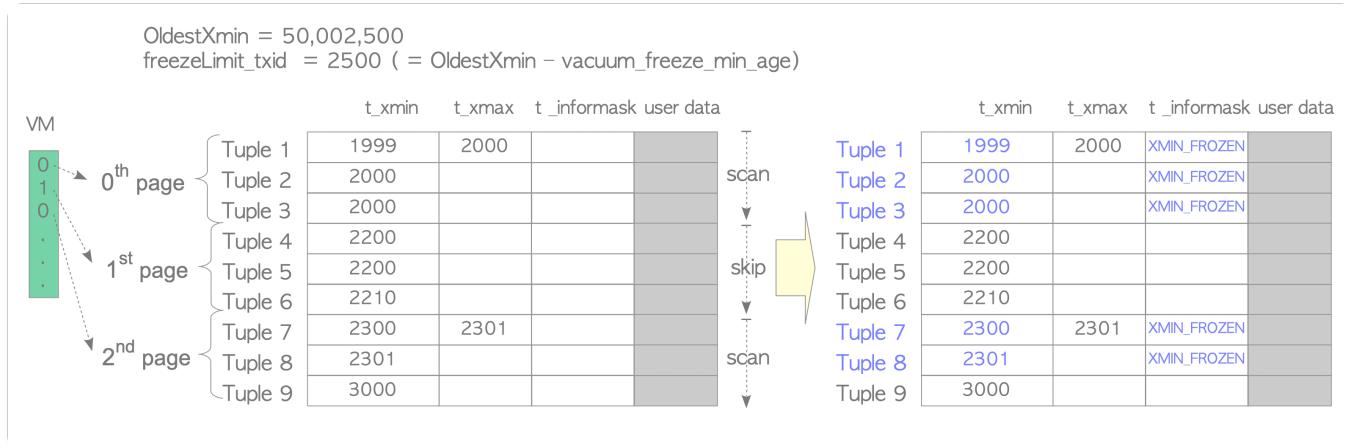
The `freezeLimit_txid` is defined as follows:

$$\text{freezeLimit\_txid} = (\text{OldestXmin} - \text{vacuum\_freeze\_min\_age})$$

where `OldestXmin` is the oldest txid among currently running transactions. For example, if three transactions (txids 100, 101, and 102) are running when the VACUUM command is executed, `OldestXmin` is 100. If no other transactions exist, `OldestXmin` is the txid that executes this VACUUM command. Here, `vacuum_freeze_min_age` is a configuration parameter (the default is 50,000,000).

Figure 6.3 shows a specific example. Here, Table\_1 consists of three pages, and each page has three tuples. When the VACUUM command is executed, the current txid is 50,002,500 and there are no other transactions. In this case, `OldestXmin` is 50,002,500; thus, the `freezeLimit_txid` is 2500. Freeze processing is executed as follows.

**Fig. 6.3. Freezing tuples in lazy mode.**



0<sup>th</sup> page:

Three tuples are frozen because all `t_xmin` values are less than the `freezeLimit_txid`. In addition, Tuple\_1 is removed in this vacuum process due to a dead tuple.

1<sup>st</sup> page:

This page is skipped by referring to the VM.

2<sup>nd</sup> page:

Tuple\_7 and Tuple\_8 are frozen; Tuple\_7 is removed.

Before completing the vacuum process, the statistics related to vacuuming are updated, e.g. pg\_stat\_all\_tables' n\_live\_tup, n\_dead\_tup, last\_vacuum, vacuum\_count, etc.

As shown in the above example, the lazy mode might not be able to freeze tuples completely because it can skip pages.

### 6.3.2. Eager Mode

The eager mode compensates for the defect of the lazy mode. It scans all pages to inspect all tuples in tables, updates relevant system catalogs, and removes unnecessary files and pages of the clog if possible.

The eager mode is performed when the following condition is satisfied:

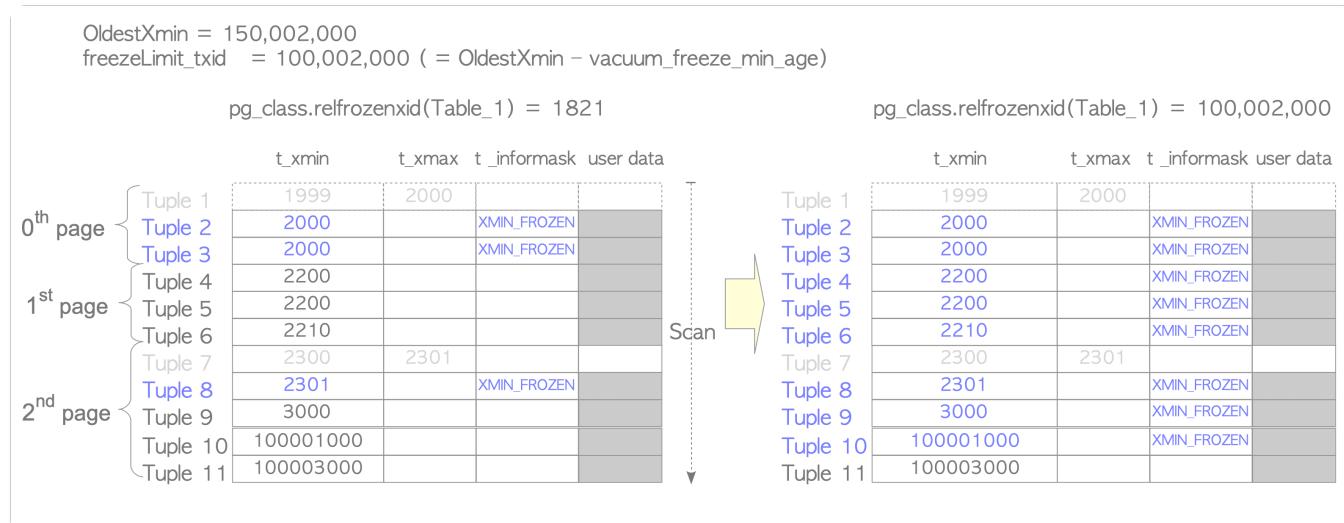
```
pg_database.datfrozenxid < (OldestXmin – vacuum_freeze_table_age)
```

In the condition above, *pg\_database.datfrozenxid* represents the columns of the pg\_database system catalog and holds the oldest frozen txid for each database. Details are described later; therefore, we assume that the value of all pg\_database.datfrozenxid are 1821 (which is the initial value just after installation of a new database cluster in version 9.5). Vacuum\_freeze\_table\_age is a configuration parameter (the default is 150,000,000).

Figure 6.4 shows a specific example. In Table\_1, both Tuple\_1 and Tuple\_7 have been removed. Tuple\_10 and Tuple\_11 have been inserted into the 2nd page. When the VACUUM command is executed, the current txid is 150,002,000, and there are no other transactions. Thus, OldestXmin is 150,002,000 and the freezeLimit txid is 100,002,000. In this case, the above condition is satisfied because '1821 < (150002000 – 150000000)'; therefore, the freeze processing performs in eager mode as follows.

(Note that this is the behavior of versions 9.5 or earlier; the latest behavior is described in Section 6.3.3.)

**Fig. 6.4. Freezing old tuples in eager mode (versions 9.5 or earlier).**



0<sup>th</sup> page:

Tuple\_2 and Tuple\_3 have been checked even though all tuples have been frozen.

1<sup>st</sup> page:

Three tuples in this page have been frozen because all t\_xmin values are less than the freezeLimit txid. Note that this page is skipped in lazy mode.

2<sup>nd</sup> page:

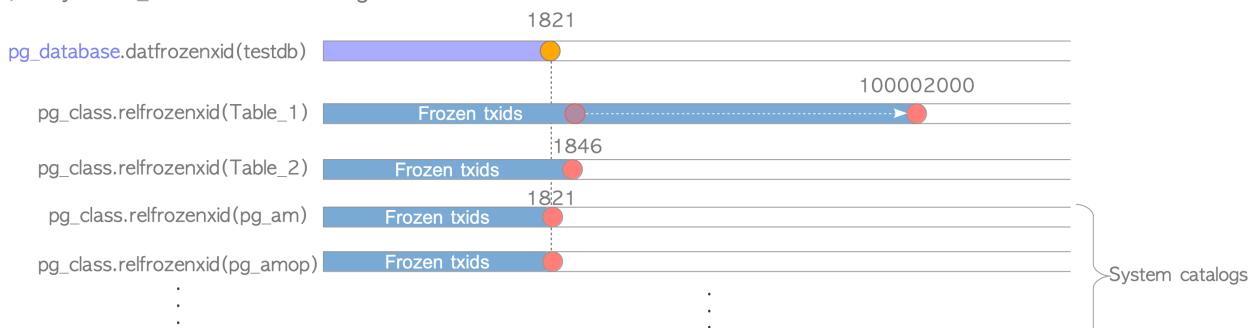
Tuple\_10 has been frozen. Tuple\_11 has not.

After freezing each table, the pg\_class.relfrozenxid of the target table is updated. The pg\_class is a system catalog, and each pg\_class.relfrozenxid column holds the latest frozen xid of the corresponding table. In this example, Table\_1's pg\_class.relfrozenxid is updated to the current freezeLimit txid (i.e. 100,002,000), which means that all tuples whose t\_xmin is less than 100,002,000 in Table\_1 are frozen.

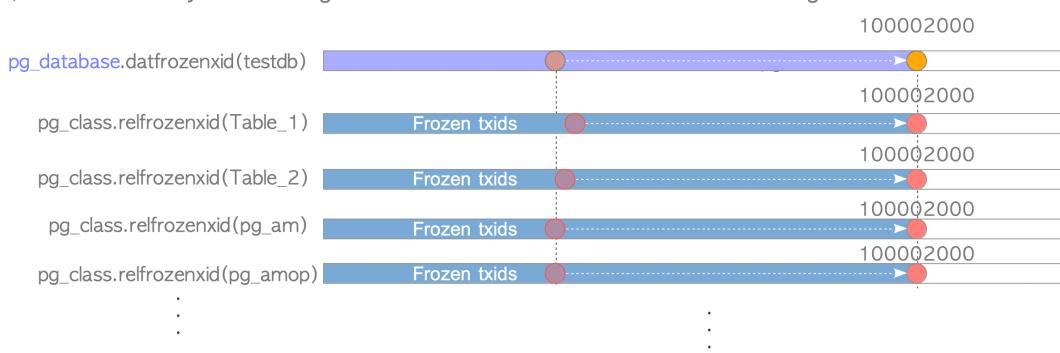
Before completing the vacuum process, pg\_database.datfrozenxid is updated if necessary. Each pg\_database.datfrozenxid column holds the minimum pg\_class.relfrozenxid in the corresponding database. For example, if only Table\_1 is frozen in eager mode, the pg\_database.datfrozenxid of this database is not updated because the pg\_class.relfrozenxid of other relations (both other tables and system catalogs that can be seen from the current database) have not been changed (Fig. 6.5(1)). If all relations in the current database are frozen in eager mode, the pg\_database.datfrozenxid of the database is updated because all relations' pg\_class.relfrozenxid for this database are updated to the current freezeLimit txid (Fig. 6.5(2)).

**Fig. 6.5. Relationship between pg\_database.datfrozenxid and pg\_class.relfrozenxid(s).**

(1) Only Table\_1 is frozen in the eager mode



(2) All tables and system catalogs in the database "testdb" are frozen in the eager mode



## 🎓 How to show pg\_class.relfrozenxid and pg\_database.datfrozenxid

In the following, the first query shows the relfrozenxids of all visible relations in the 'testdb' database, and the second query shows the pg\_database.datfrozenxid of the 'testdb' database.

```
testdb=# VACUUM table_1;
VACUUM

testdb=# SELECT n.nspname AS "Schema", c.relname AS "Name", c.relfrozenxid
```

```

FROM pg_catalog.pg_class c
LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkid IN ('r', '')
    AND n.nspname <> 'information_schema' AND n.nspname !~ '^pg_toast'
    AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY c.relfrozenxid::text::bigint DESC;

```

Schema	Name	relfrozenxid
public	table_1	100002000
public	table_2	1846
pg_catalog	pg_database	1827
pg_catalog	pg_user_mapping	1821
pg_catalog	pg_largeobject	1821
...		
pg_catalog	pg_transform	1821

(57 rows)

```

testdb=# SELECT datname, datfrozenxid FROM pg_database WHERE datname = 'testdb';

```

datname	datfrozenxid
testdb	1821

(1 row)

## ❶ FREEZE option

The VACUUM command with the FREEZE option forces all txids in the specified tables to be frozen. This is performed in eager mode, but the freezeLimit is set to OldestXmin (not 'OldestXmin - vacuum\_freeze\_min\_age'). For example, when the VACUUM FULL command is executed by txid 5000 and there are no other running transactions, OldestXmin is set to 5000 and txids that are less than 5000 are frozen.

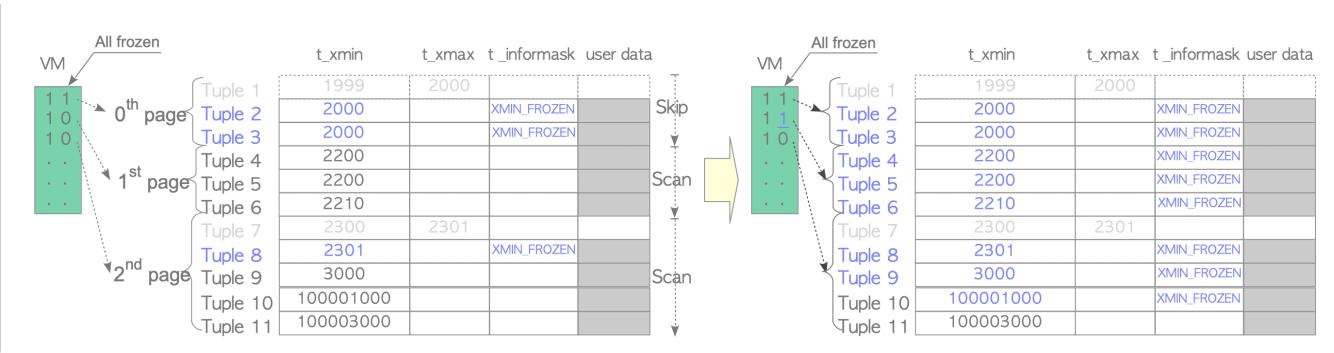
### 6.3.3. Improving Freeze Processing in Eager Mode

The eager mode in version 9.5 or earlier versions is not efficient because always scans all pages. For instance, in the example of Section 6.3.2, the 0th page is scanned even though all tuples in its page are frozen.

To deal with this issue, the VM and freeze process have been improved in version 9.6. As mentioned in Section 6.2.1, the new VM has information about whether all tuples are frozen in each page. When freeze processing is executed in eager mode, pages that contain only frozen tuples can be skipped.

Figure 6.6 shows an example. When freezing this table, the 0th page is skipped by referring to the VM's information. After freezing the 1st page, the associated VM information is updated because all tuples of this page have been frozen.

**Fig. 6.6. Freezing old tuples in eager mode (versions 9.6 or later).**

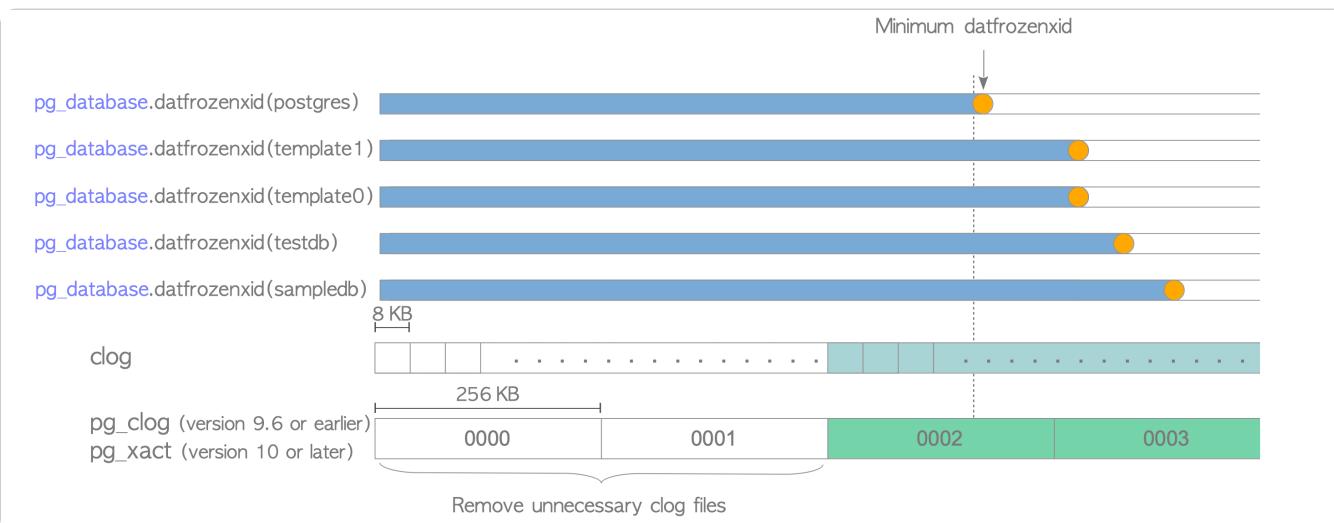


## **6.4. Removing Unnecessary Clog Files**

The clog, described in Section 5.4, stores transaction states. When `pg_database.datfrozenxid` is updated, PostgreSQL attempts to remove unnecessary clog files. Note that corresponding clog pages are also removed.

Figure 6.7 shows an example. If the minimum pg\_database.datfrozenxid is contained in the clog file '0002', the older files ('0000' and '0001') can be removed because all transactions stored in those files can be treated as frozen txids in the whole database cluster.

**Fig. 6.7. Removing unnecessary clog files and pages.**



grad pg database.datfrozenxid and the clog file

The following shows the actual output of pg\_database.datfrozenxid and the clog files:

```
$ psql testdb -c "SELECT datname, datfrozenxid FROM pg_database"
   datname | datfrozenxid
-----
template1 |      7308883
template0 |      7556347
postgres  |      7339732
testdb    |      7506298
(4 rows)

$ ls -la -h data/pg_xact/          # In versions 9.6 or earlier, "ls -la -h data/pg_clog/"
total 316K
drwx----- 2 postgres postgres  28 Dec 29 17:15 .
drwx----- 20 postgres postgres 4.0K Dec 29 17:13 ..
-rw----- 1 postgres postgres 256K Dec 29 17:15 0006
-rw----- 1 postgres postgres 56K Dec 29 17:15 0007
```

## 6.5. Autovacuum Daemon

Vacuum processing has been automated with the autovacuum daemon, making the operation of PostgreSQL extremely easy.

The autovacuum daemon periodically invokes several autovacuum\_worker processes. By default, it wakes every 1 minute (defined by `autovacuum_naptime`) and invokes three workers (defined by `autovacuum_max_workers`).

The autovacuum workers invoked by the autovacuum deamon perform vacuum processing concurrently for respective tables, gradually and with minimal impact on database activity.

### 6.5.1. Conditions for autovacuum to run

The autovacuum process runs for a target table if any of the following conditions are satisfied:

1. The current txid precedes the following expression:

```
realfrozenxid + autovacuum_freeze_max_age,
```

where `realfrozenxid` is the `realfrozenxid` value of the target table that is defind in the `pg_class`, and `autovacuum_freeze_max_age` (the default is 200,000,000) is a configuration parameter.

If this condition is satisfied, the autovacuum process runs for the target table to perform freeze processing.

2. The number of dead tuples is greater than the following expression:

```
autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor × reltuples,
```

where `autovacuum_vacuum_threshold` (the default is 50) and `autovacuum_vacuum_scale_factor` (the default is 0.2) are configuration parameters, `reltuples` is the number of tuples in the target table.

For example, if the target table has 10,000 tuples and 2,100 dead tuples, the autovacuum process runs for the target table since  $2100 > 50 + 0.2 \times 10000$ .

3. The number of inserted tuples in the target table is greater than the following expression:

```
autovacuum_vacuum_insert_threshold + autovacuum_vacuum_insert_scale_factor × reltuples,
```

where `autovacuum_vacuum_insert_threshold` (the default is 1000) and `autovacuum_vacuum_insert_scale_factor` (the default is 0.2) are configuration parameters, `reltuples` is the number of tuples in the target table.

For example, if the target table has 10,000 tuples and 3,010 inserted tuples, the autovacuum process runs for the target table since  $3010 > 1000 + 0.2 \times 10000$ .

This condition has been added since version 13.

In addition, if the following condition is satisfied for the target table, the autovacuum process will also perform analyze processing.

```
mod_since_analyze > autovacuum_analyze_threshold + autovacuum_analyze_scale_factor × reltuples,
```

where `mod_since_analyze` is the number of modified tuples (by INSERT, DELETE or UPDATE) since the previous analyze processing, `autovacuum_analyze_threshold` (the default is 50) and `autovacuum_analyze_scale_factor` (the default is 0.1) are configuration parameters, `reltuples` is the number of tuples in the target table.

For example, if the target table has 10,000 tuples and 1,100 modified tuples since the previous analyze processing, the autovacuum process will run since  $1100 > 50 + 0.1 \times 10000$ .



The function `relation_needs_vacanalyze()` determines whether target tables need to be vacuumed or analyzed.

## 6.5.2. Maintenance tips

As frequently mentioned, table bloat is one of the most annoying things in managing PostgreSQL. Several things can cause that problem, and Autovacuum is one of them.

The autovacuum runs when the number of dead tuples is greater than: 250 for 1,000 relations, 20,050 for 100,000 relations, and 20,000,050 for 100,000,000 relations. It is clear from these examples that the more tuples a table has, the less often autovacuum runs.

A good known tip is to reduce the **autovacuum\_vacuum\_scale\_factor** value. In fact, the default of `autovacuum_vacuum_scale_factor` (0.2) is too large for big tables.

PostgreSQL can set an appropriate `autovacuum_vacuum_scale_factor` in each table using `ALTER TABLE` command. For example, I show how to set the new value of `autovacuum_vacuum_scale_factor` for the table `pgbench_accounts`.

```
postgres=# ALTER TABLE pgbench_accounts SET (autovacuum_vacuum_scale_factor = 0.05);  
ALTER TABLE
```

If you need that Autovacuum runs for the target tables without depending on the number of their tuples, you can also do it.

For example, assume that you need that Autovacuum processing whenever the number of dead tuples reaches 10,000. In this case, by setting the following storage parameters for the table, the Autovacuum process will perform the vacuum processing each time it reaches 10,000:

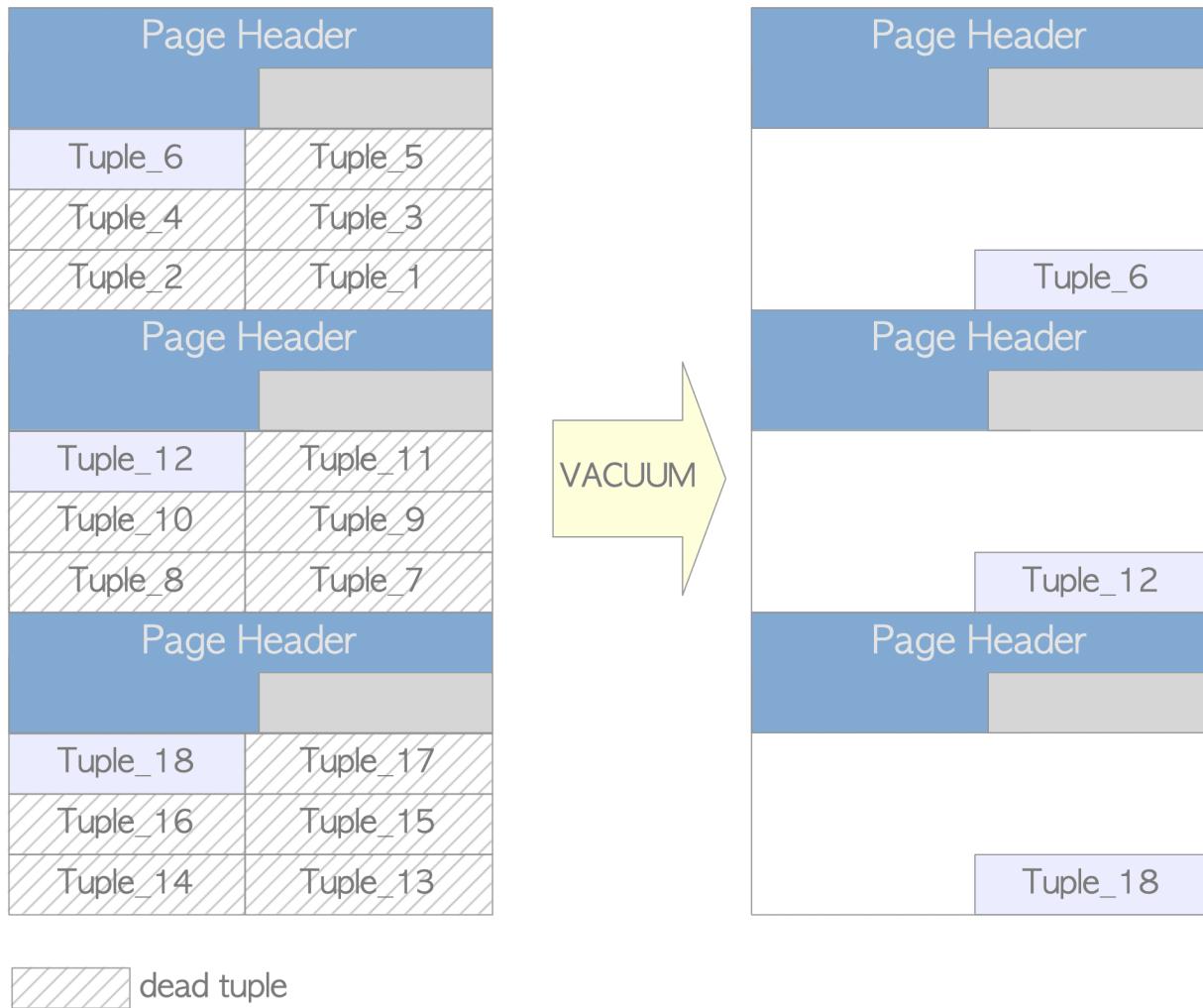
```
postgres=# ALTER TABLE pgbench_accounts SET (autovacuum_vacuum_threshold = 10000);  
ALTER TABLE  
postgres=# ALTER TABLE pgbench_accounts SET (autovacuum_vacuum_scale_factor = 0.0);  
ALTER TABLE
```

## 6.6. Full VACUUM

Although Concurrent VACUUM is essential for operation, it is not sufficient. For example, it cannot reduce the size of a table even if many dead tuples are removed.

Figure 6.8 shows an extreme example. Suppose that a table consists of three pages, and each page contains six tuples. The following DELETE command is executed to remove tuples, and the VACUUM command is executed to remove dead tuples:

**Fig. 6.8. An example showing the disadvantages of (concurrent) VACUUM.**

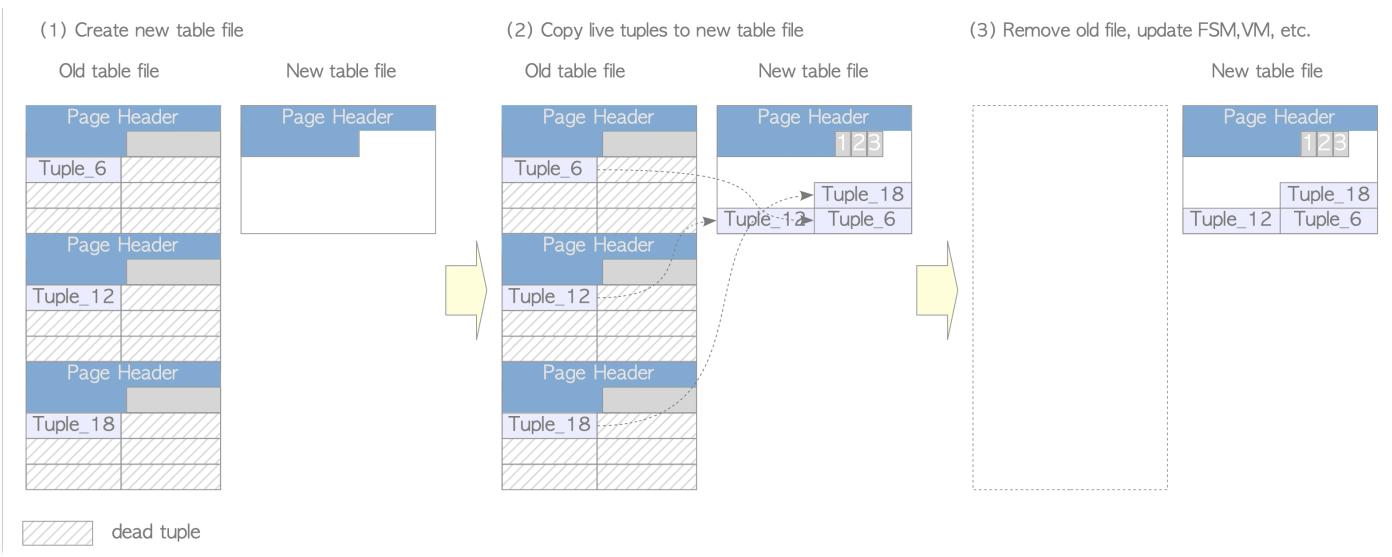


```
testdb=# DELETE FROM tbl WHERE id % 6 != 0;
testdb=# VACUUM tbl;
```

The dead tuples are removed; but the table size is not reduced. This is both a waste of disk space and has a negative impact on database performance. For instance, in the above example, when three tuples in the table are read, three pages must be loaded from disk.

To deal with this situation, PostgreSQL provides the Full VACUUM mode. Figure 6.9 shows an outline of this mode.

**Fig. 6.9. Outline of Full VACUUM mode.**



### [1] Create new table file: Fig. 6.9(1)

When the VACUUM FULL command is executed for a table, PostgreSQL first acquires the AccessExclusiveLock lock for the table and creates a new table file whose size is 8 KB. The AccessExclusiveLock lock prevents other users from accessing the table.

### [2] Copy live tuples to the new table: Fig. 6.9(2)

PostgreSQL copies only live tuples within the old table file to the new table.

### [3] Remove the old file, rebuild indexes, and update the statistics, FSM, and VM: Fig. 6.9(3)

After copying all live tuples, PostgreSQL removes the old file, rebuilds all associated table indexes, updates both the FSM and VM of this table, and updates associated statistics and system catalogs.

The pseudocode of the Full VACUUM is shown in below:

### </> Pseudocode: Full VACUUM

```

(1) FOR each table
(2)   Acquire AccessExclusiveLock lock for the table
(3)   Create a new table file
(4)   FOR each live tuple in the old table
(5)     Copy the live tuple to the new table file
(6)     Freeze the tuple IF necessary
  END FOR
(7)   Remove the old table file
(8)   Rebuild all indexes
(9)   Update FSM and VM
(10)  Update statistics
    Release AccessExclusiveLock lock
  END FOR
(11) Remove unnecessary clog files and pages if possible

```

Two points should be considered when using the VACUUM FULL command.

1. Nobody can access(read/write) the table when Full VACUUM is processing.
2. At most twice the disk space of the table is used temporarily; therefore, it is necessary to check the remaining disk capacity when a huge table is processed.

### 🎓 When should I do VACUUM FULL?

There is unfortunately no one-size-fits-all answer to the question of when to execute VACUUM FULL. However, the extension pg\_freespacemap can give you some good suggestions.

The following query shows the average freespace ratio of the table you want to know.

```
testdb=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION

testdb=# SELECT count(*) as "number of pages",
    pg_size.pretty(cast(avg(avail) as bigint)) as "Av. freespace size",
    round(100 * avg(avail)/8192 ,2) as "Av. freespace ratio"
    FROM pg_freespace('accounts');
number of pages | Av. freespace size | Av. freespace ratio
-----+-----+-----+
      1640 |   99 bytes          |       1.21
(1 row)
```

As the result above, You can find that there are few free spaces (1.21% free space).

If you delete almost tuples and execute VACUUM command, you can find that almost pages are empty (86.97% free space), but the number of pages remains the same. In other words, the table file has not been compacted.

```
testdb=# DELETE FROM accounts WHERE aid %10 != 0 OR aid < 100;
DELETE 90009

testdb=# VACUUM accounts;
VACUUM

testdb=# SELECT count(*) as "number of pages",
    pg_size.pretty(cast(avg(avail) as bigint)) as "Av. freespace size",
    round(100 * avg(avail)/8192 ,2) as "Av. freespace ratio"
    FROM pg_freespace('accounts');
number of pages | Av. freespace size | Av. freespace ratio
-----+-----+-----+
      1640 |  7124 bytes          |      86.97
(1 row)
```

The following query inspects the freespace ratio of each page of the specified table.

```
testdb=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
    FROM pg_freespace('accounts');
blkno | avail | freespace ratio
-----+-----+-----+
      0 |  7904 |      96.00
      1 |  7520 |      91.00
      2 |  7136 |      87.00
      3 |  7136 |      87.00
      4 |  7136 |      87.00
      5 |  7136 |      87.00
...
...
```

If you run VACUUM FULL in this situation, you will see that the table file has been compacted.

```
testdb=# VACUUM FULL accounts;
VACUUM

testdb=# SELECT count(*) as "number of blocks",
    pg_size.pretty(cast(avg(avail) as bigint)) as "Av. freespace size",
    round(100 * avg(avail)/8192 ,2) as "Av. freespace ratio"
    FROM pg_freespace('accounts');
number of pages | Av. freespace size | Av. freespace ratio
-----+-----+-----+
      164 |   0 bytes           |       0.00
(1 row)
```

# Chapter 7

## Heap Only Tuple and Index-Only Scans

This chapter describes two features related to the index scan: the heap only tuple and index-only scans.

### 7.1. Heap Only Tuple (HOT)

The HOT was implemented in version 8.3 to effectively use the pages of both index and table when the updated row is stored in the same table page that stores the old row. HOT also reduces the need for VACUUM processing.

Since the details of HOT are described in the README.HOT file in the source code directory, this chapter only provides a brief introduction to HOT. Section 7.1.1 first describes how to update a row without HOT to clarify the issues that HOT resolves. Section 7.1.2 then describes how HOT works.

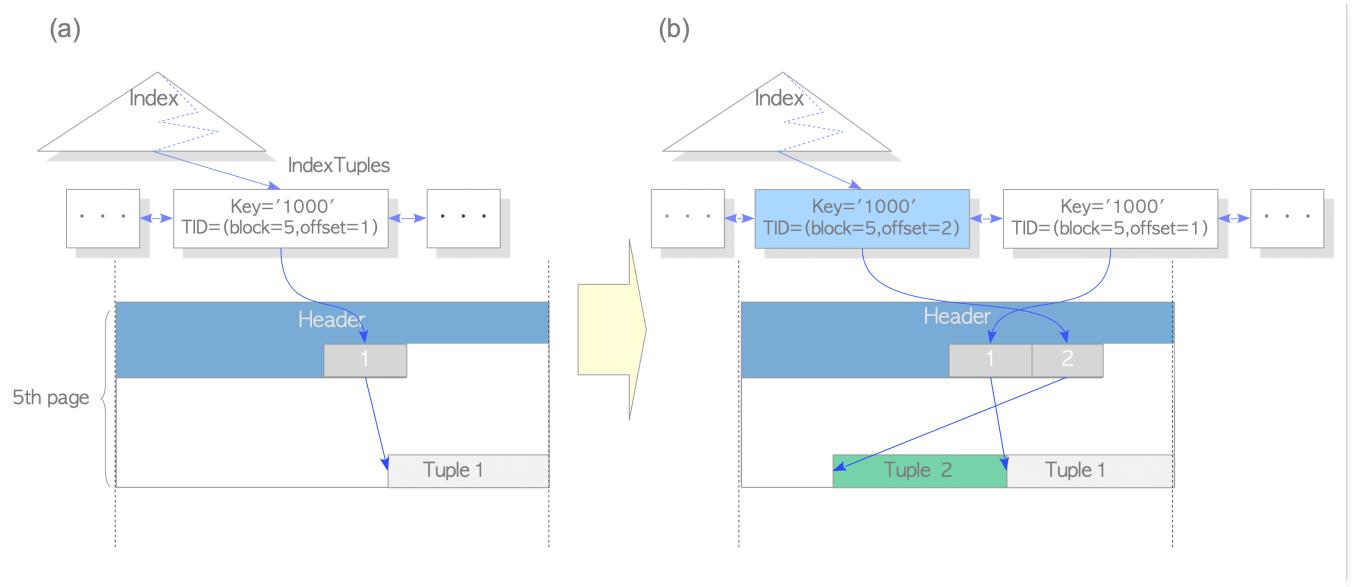
#### 7.1.1. Update a Row Without HOT

Assume that the table 'tbl' has two columns: 'id' and 'data'; 'id' is the primary key of 'tbl'.

```
testdb=# \d tbl
      Table "public.tbl"
 Column | Type    | Collation | Nullable | Default
-----+---------+-----------+----------+
 id   | integer |           | not null |
 data | text    |           |           |
Indexes:
 "tbl_pkey" PRIMARY KEY, btree (id)
```

The table 'tbl' has 1000 tuples; the last tuple, whose id is 1000, is stored in the 5th page of the table. The last tuple is pointed to by the corresponding index tuple, whose key is 1000 and whose tid is '(5, 1)'. See Fig. 7.1(a).

**Fig. 7.1. Update a row without HOT**



We consider how the last tuple is updated without HOT.

```
testdb=# UPDATE tbl SET data = 'B' WHERE id = 1000;
```

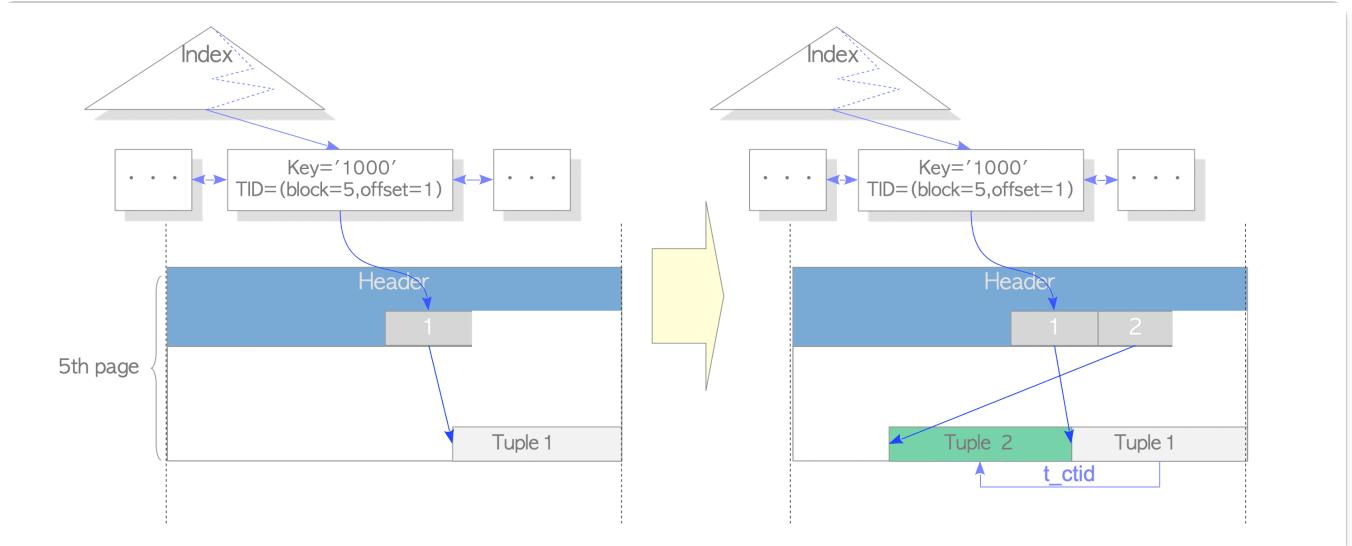
In this case, PostgreSQL inserts not only the new table tuple but also the new index tuple into the index page. See Fig. 7.1(b).

The inserting of the index tuples consumes the index page space, and both the inserting and vacuuming costs of the index tuples are high. HOT reduces the impact of these issues.

## 7.1.2. How HOT Performs

When a row is updated with HOT, if the updated row will be stored in the same table page that stores the old row, PostgreSQL does not insert the corresponding index tuple and sets the HEAP\_HOT\_UPDATED bit and the HEAP\_ONLY\_TUPLE bit to the t\_informask2 fields of the old tuple and the new tuple, respectively. See Figs. 7.2 and 7.3.

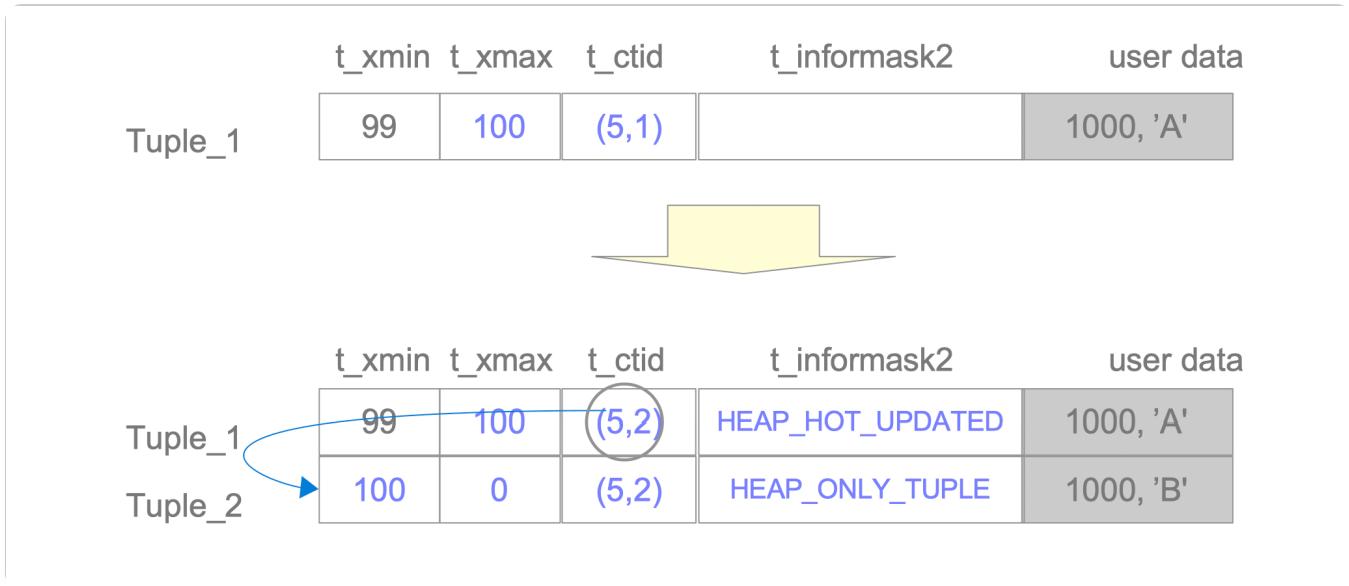
**Fig. 7.2. Update a row with HOT**



For example, in this case, 'Tuple\_1' and 'Tuple\_2' are set to the HEAP\_HOT\_UPDATED bit and the HEAP\_ONLY\_TUPLE bit, respectively.

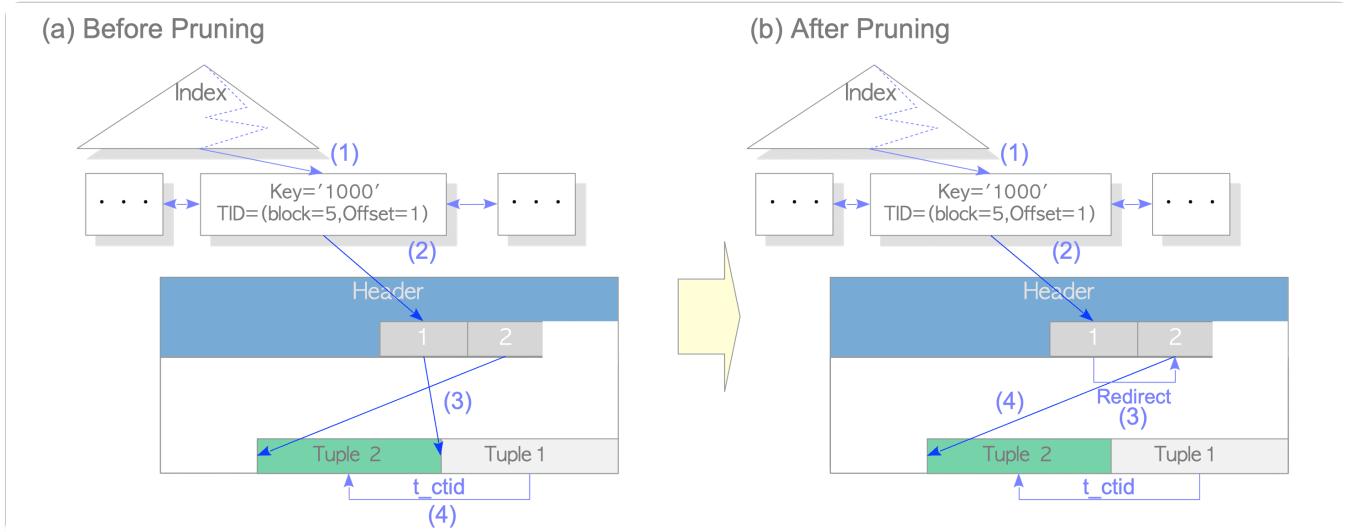
In addition, the HEAP\_HOT\_UPDATED and the HEAP\_ONLY\_TUPLE bits are used regardless of the *pruning* and the *defragmentation* processes, which are described in the following, are executed.

**Fig. 7.3. HEAP\_HOT\_UPDATED and HEAP\_ONLY\_TUPLE bits**



In the following, a description of how PostgreSQL accesses the updated tuples using the index scan immediately after updating the tuples with HOT is given. See Fig. 7.4(a).

**Fig. 7.4. Pruning of the line pointers**



- (1) Find the index tuple that points to the target tuple.
- (2) Access the line pointer '[1]' that is pointed to by the index tuple.
- (3) Read 'Tuple\_1'.
- (4) Read 'Tuple\_2' via the t\_ctid of 'Tuple\_1'.

In this case, PostgreSQL reads two tuples, 'Tuple\_1' and 'Tuple\_2', and decides which is visible using the concurrency control mechanism described in Chapter 5.

However, a problem arises if the dead tuples in the table pages are removed. For example, in Fig. 7.4(a), if 'Tuple\_1' is removed since it is a dead tuple, 'Tuple\_2' cannot be accessed from the index.

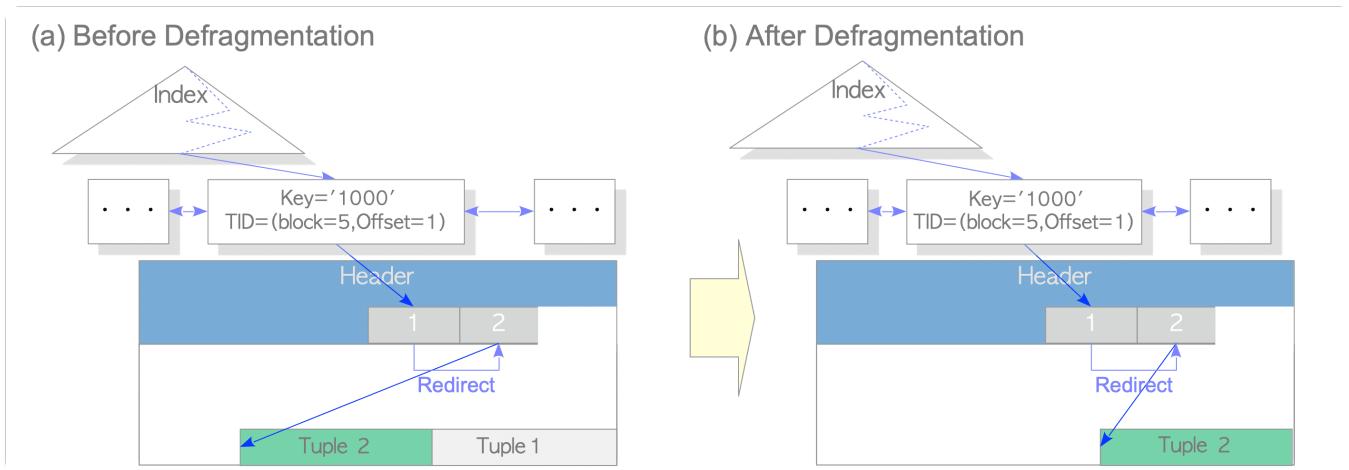
To resolve this problem, at an appropriate time, PostgreSQL redirects the line pointer that points to the old tuple to the line pointer that points to the new tuple. In PostgreSQL, this processing is called **pruning**. Fig. 7.4(b) depicts how PostgreSQL accesses the updated tuples after pruning.

- (1) Find the index tuple.
- (2) Access the line pointer '[1]' that is pointed to by the index tuple.
- (3) Access the line pointer '[2]' that points to 'Tuple\_2' via the redirected line pointer.
- (4) Read 'Tuple\_2' that is pointed to by the line pointer '[2]'.

The pruning processing will be executed, if possible, when a SQL command is executed such as SELECT, UPDATE, INSERT and DELETE. The exact execution timing is not described in this chapter because it is very complicated. The details are described in the [README.HOT](#) file.

PostgreSQL removes dead tuples if possible, as in the pruning process, at an appropriate time. In the document of PostgreSQL, this processing is called **defragmentation**. Fig. 7.5 depicts the defragmentation by HOT.

**Fig. 7.5. Defragmentation of the dead tuples**



Note that the cost of defragmentation is less than the cost of normal VACUUM processing because defragmentation does not involve removing the index tuples.

Thus, using HOT reduces the consumption of both indexes and tables of pages; this also reduces the number of tuples that the VACUUM processing has to process. Therefore, HOT has a positive influence on performance because it eventually reduces the number of insertions of the index tuples by updating and the necessity of VACUUM processing.

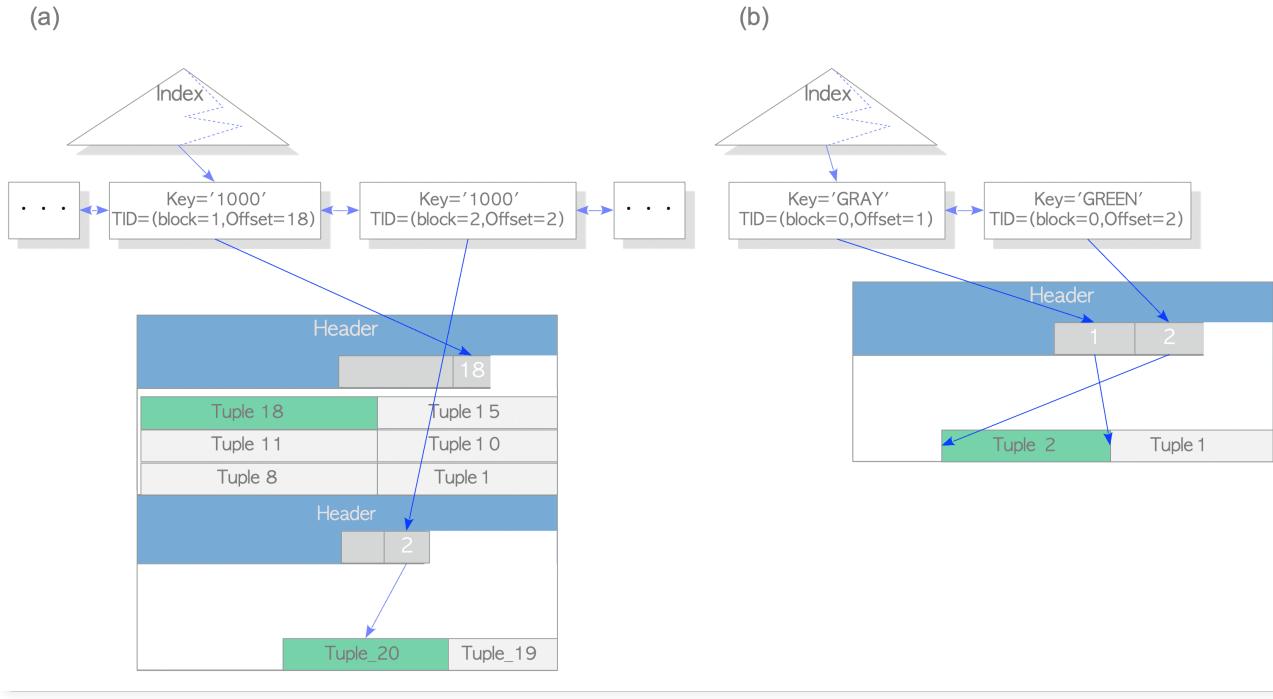
## **❶ The Cases in which HOT is not available**

To clearly understand how HOT performs, I will describe the cases in which HOT is not available.

When the updated tuple is stored in a different page from the page that stores the old tuple, the index tuple that points to the tuple must also be inserted in the index page. See Fig. 7.6(a).

When the key value of the index tuple is updated, a new index tuple must be inserted in the index page. See Figure 7.6(b).

**Fig. 7.6. The Cases in which HOT is not available**



### 1 The statistics related to the HOT

The pg\_stat\_all\_tables view provides a statistics value for each table. See also this extension.

## 7.2. Index-Only Scans

To reduce the I/O (input/output) cost, index-only scans (often called index-only access) directly use the index key without accessing the corresponding table pages when all of the target entries of the SELECT statement are included in the index key. This technique is provided by almost all commercial RDBMS, such as DB2 and Oracle. PostgreSQL has introduced this option since version 9.2.

In the following, using a specific example, a description of how index-only scans in PostgreSQL perform is given.

The assumptions of the example are explained below:

- Table definition

We have a table 'tbl' of which the definition is shown below:

```
testdb=# \d tbl
  Table "public.tbl"
 Column | Type   | Modifiers
-----+-----+
 id    | integer |
 name  | text    |
 data  | text    |
Indexes:
  "tbl_idx" btree (id, name)
```

- Index

The table 'tbl' has an index 'tbl\_idx', which is composed of two columns: 'id' and 'name'.

- Tuples

'tbl' has already inserted tuples.

'Tuple\_18', of which the id is '18' and name is 'Queen', is stored in the 0th page.

'Tuple\_19', of which the id is '19' and name is 'BOSTON', is stored in the 1st page.

- Visibility

All tuples in the 0th page are always visible; the tuples in the 1st page are not always visible.

Note that the visibility of each page is stored in the corresponding visibility map (VM), and the VM is described in Section 6.2.

Let us explore how PostgreSQL reads tuples when the following SELECT command is executed.

```
testdb=# SELECT id, name FROM tbl WHERE id BETWEEN 18 and 19;
 id | name
----+---
 18 | Queen
 19 | Boston
(2 rows)
```

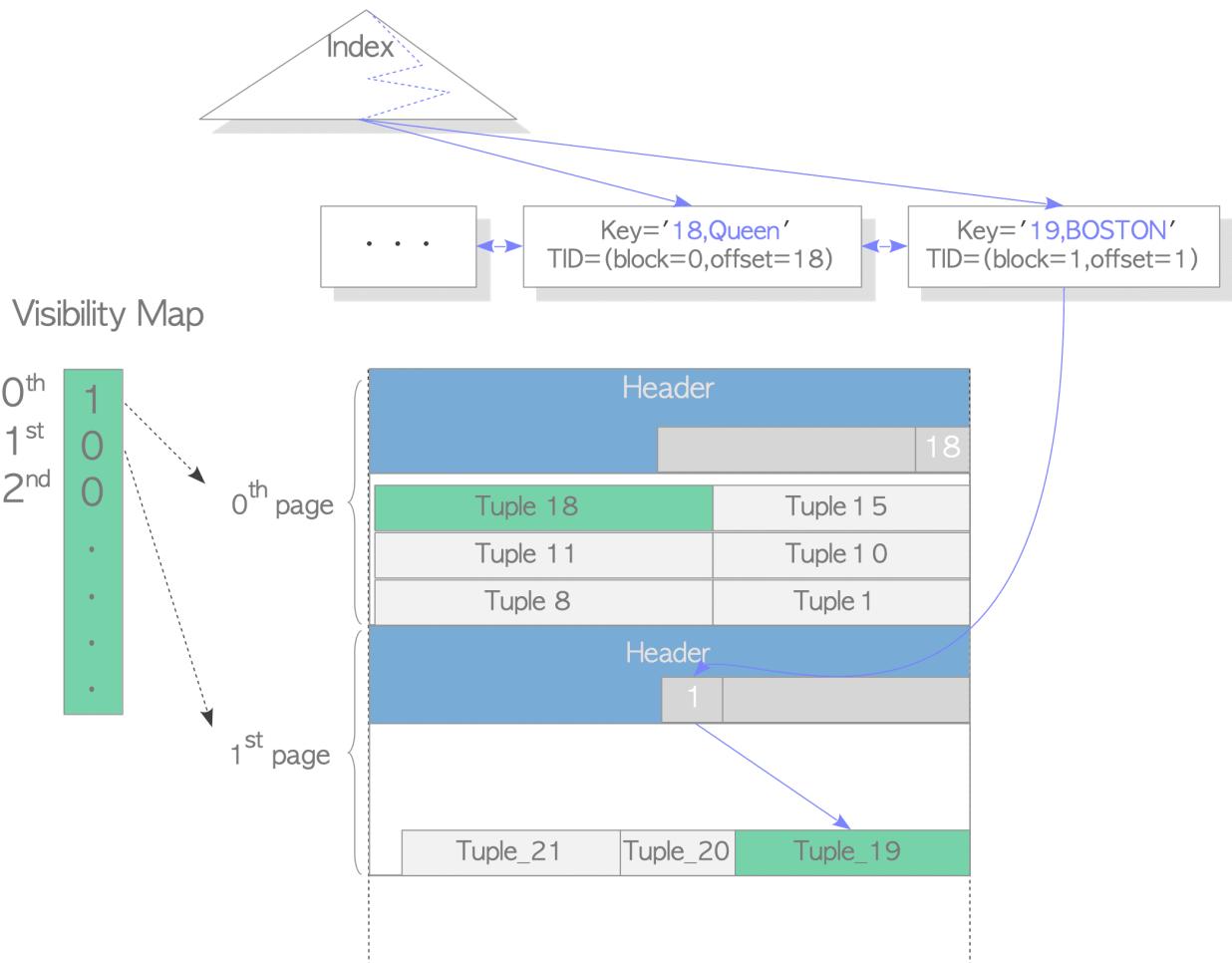
This query gets data from two columns of the table, 'id' and 'name', and the index 'tbl\_idx' is composed of these columns. Thus, it seems at first glance that accessing the table pages is not required when using index scan, because the index tuples contain the necessary data. However, in fact, PostgreSQL has to check the visibility of the tuples in principle. The index tuples do not have any information about transactions, such as the t\_xmin and t xmax of the heap tuples, which are described in Section 5.2. Therefore, PostgreSQL has to access the table data to check the visibility of the data in the index tuples. This is like putting the cart before the horse.

To avoid this dilemma, PostgreSQL uses the visibility map of the target table. If all tuples stored in a page are visible, PostgreSQL uses the key of the index tuple and does not access the table page that is pointed at from the index tuple to check its visibility. Otherwise, PostgreSQL reads the table tuple that is pointed at from the index tuple and checks the visibility of the tuple, which is the ordinary process.

In this example, 'Tuple\_18' does not need to be accessed because the 0th page that stores 'Tuple\_18' is visible. That is, all tuples in the 0th page, including 'Tuple\_18', are visible. In contrast, 'Tuple\_19' needs to be accessed to handle concurrency control because the visibility of the 1st page is not visible. See Fig. 7.7.

**Fig. 7.7. How Index-Only Scans performs**

SELECT id, key FROM tbl WHERE id BWTEEN 18 AND 19;



# Chapter 8

## Buffer Manager

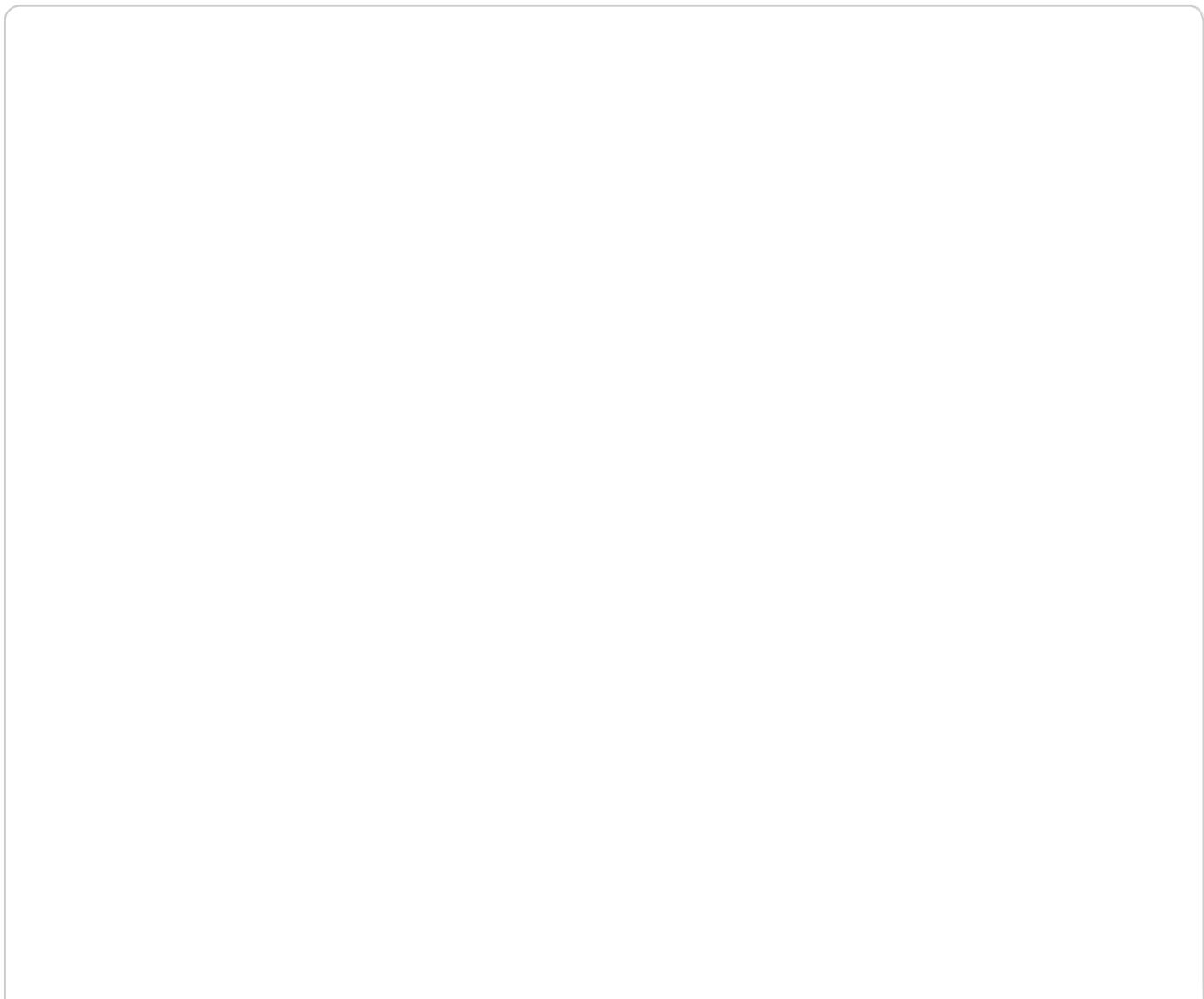
---

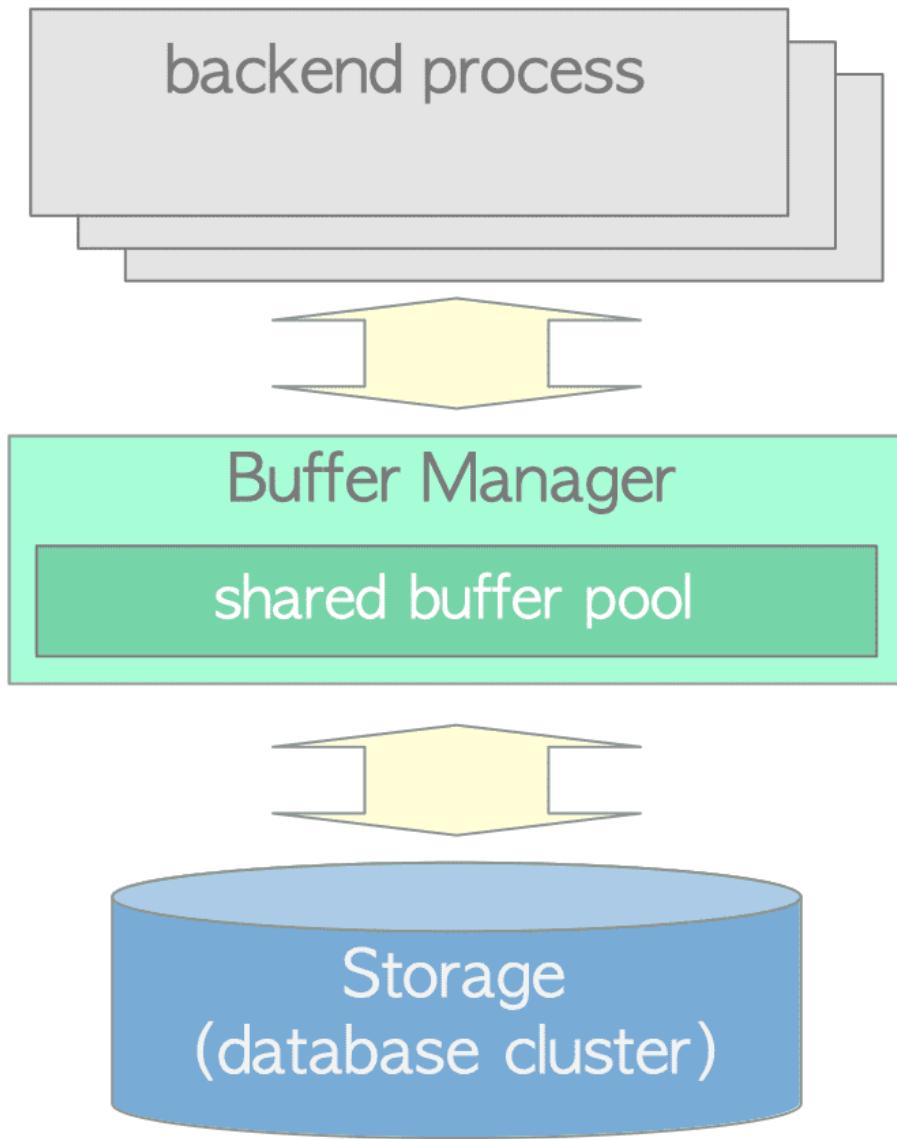
The buffer manager manages data transfers between shared memory and persistent storage, and it can have a significant impact on the performance of the DBMS. The PostgreSQL buffer manager works very efficiently.

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\_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 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.

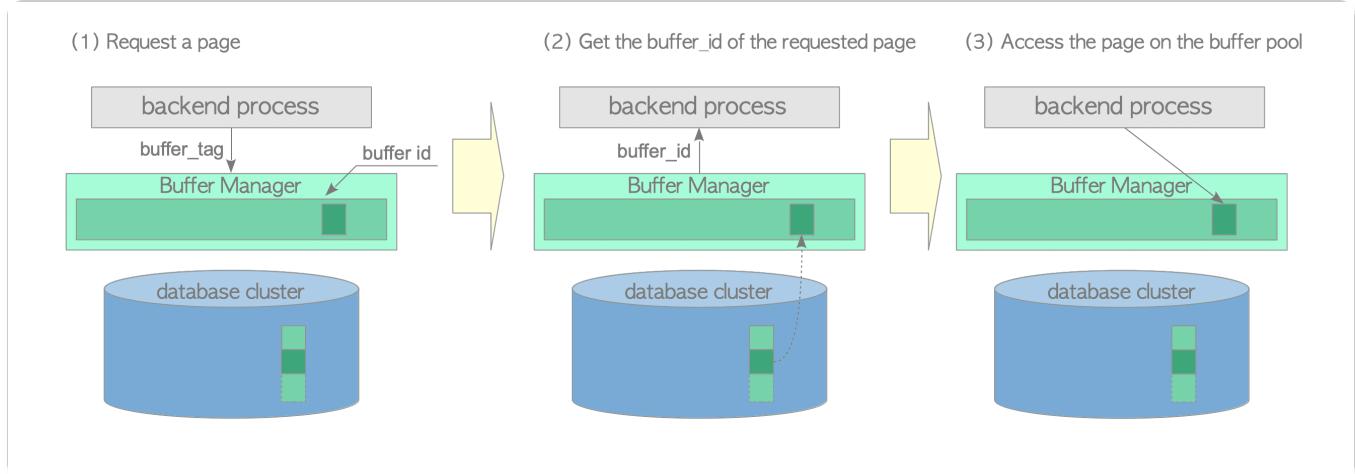
For example, the buffer\_tag '{16821, 16384, 37721, 0, 7}' identifies the page that is in the seventh block of the table whose OID and fork number are 37721 and 0, respectively. The table is contained in the database whose OID is 16384 under the tablespace whose OID is 16821.

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

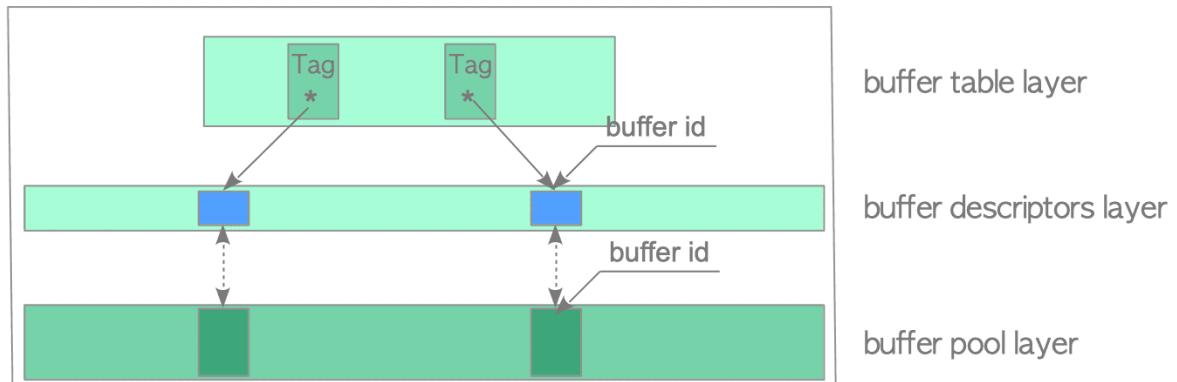
PostgreSQL versions 15 and earlier do not support direct I/O, although it has been discussed. Refer to this discussion on the pgsql-ML and this article.

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\_ids*.

- **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\_tags* of stored pages and the *buffer\_ids* of the descriptors that hold the stored pages' respective metadata.

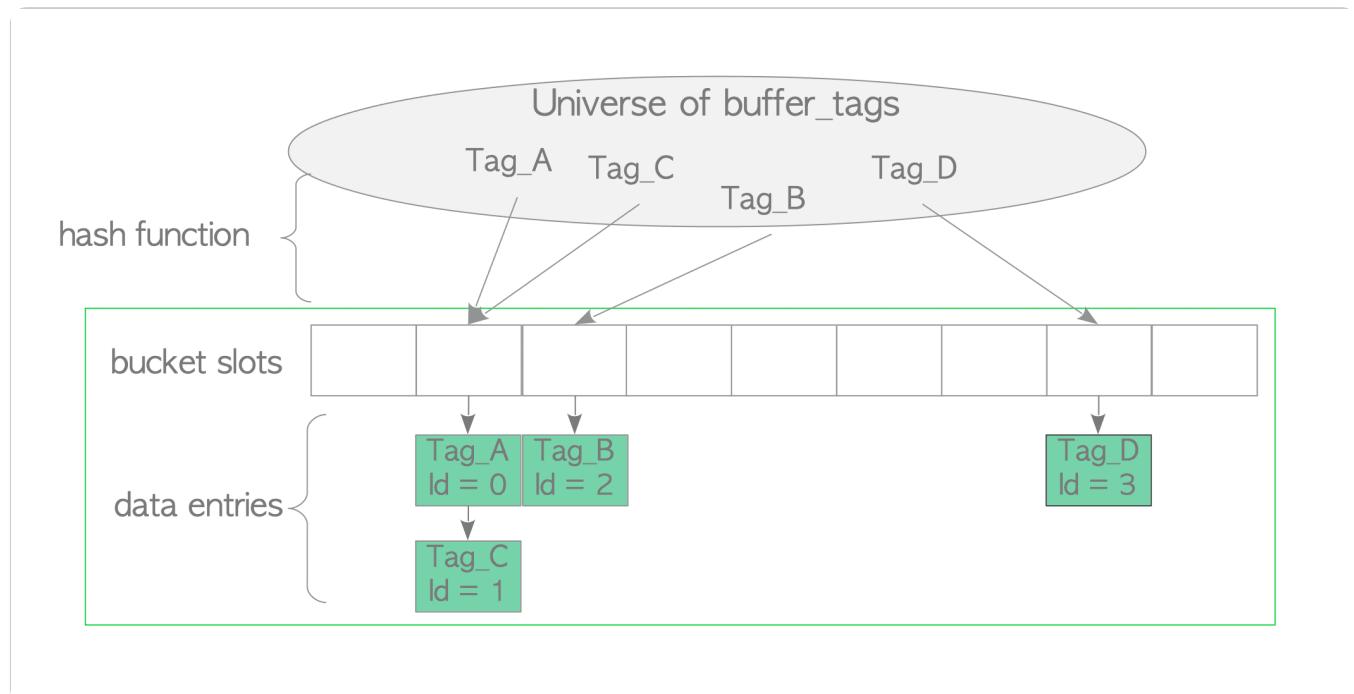
These layers are described in detail in the following subsections.

## 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 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*.

### ❶ 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 */
    BuffFlags      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  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.**

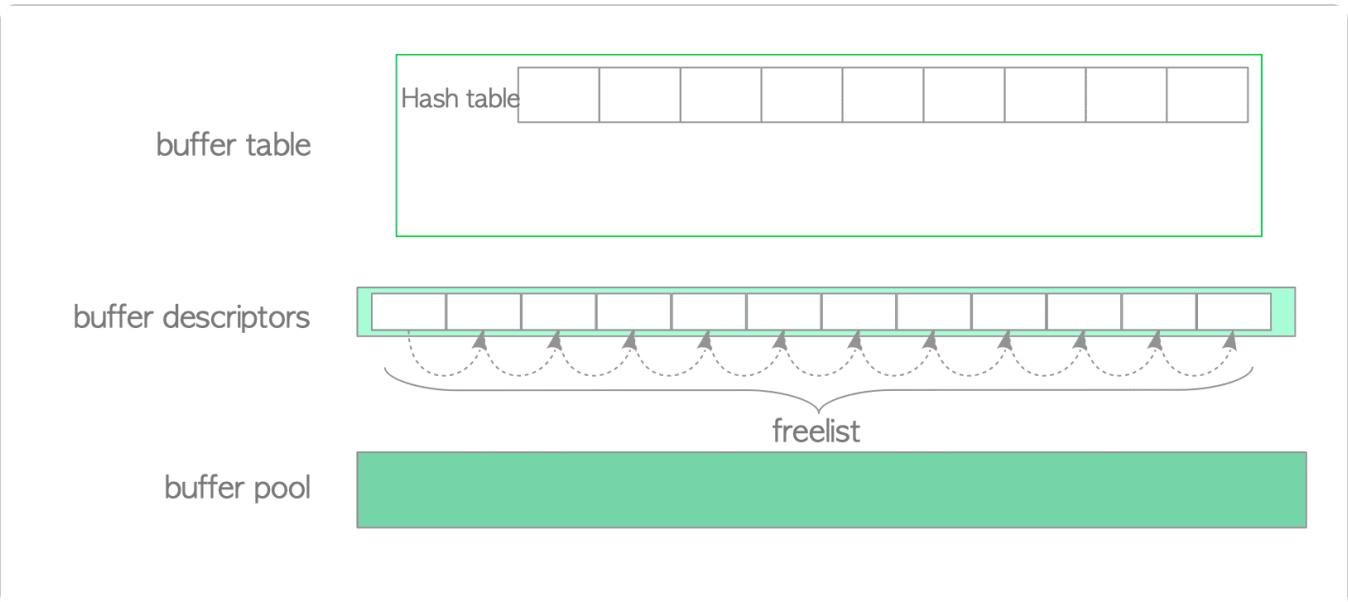
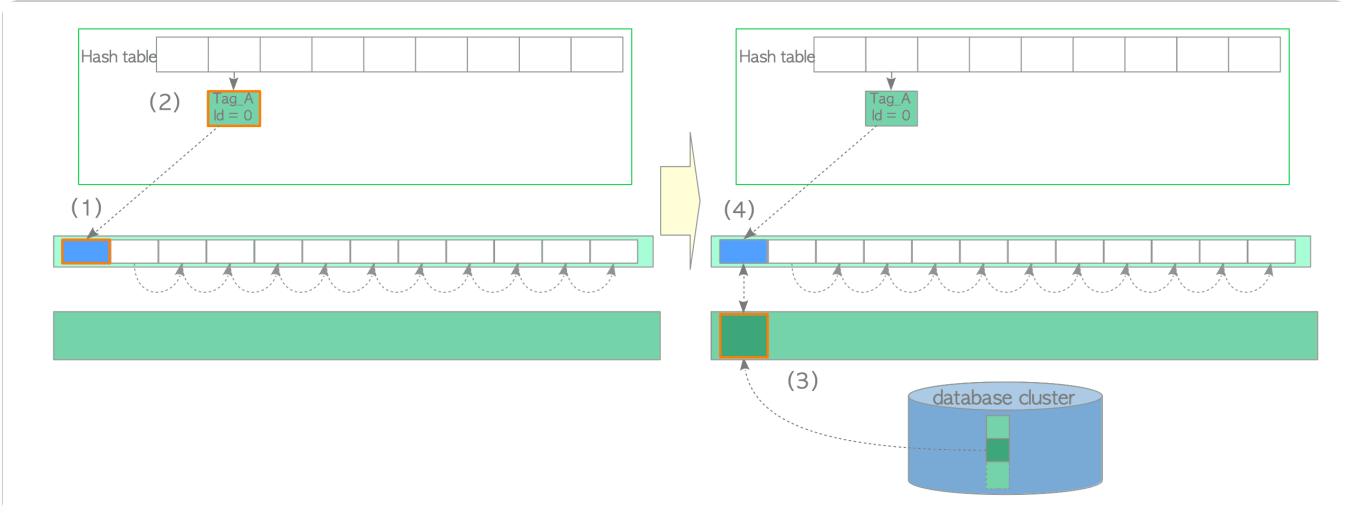


Figure 8.6 shows that how the first page is loaded.

- (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.

### ❶ 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\_ids*.

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.

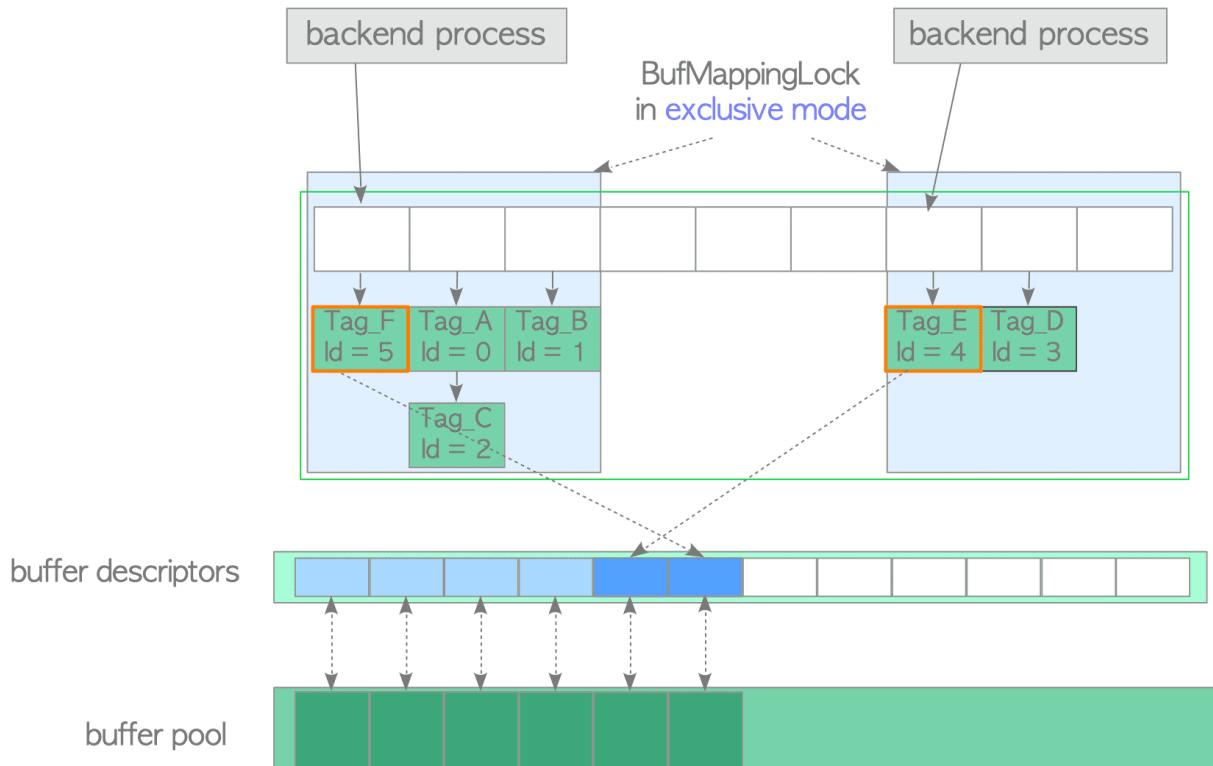
### 8.3.1. Buffer Table Locks

**BuMappingLock** 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 BuMappingLock. When inserting or deleting entries, a backend process holds an exclusive lock.

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

Figure 8.7 shows a typical example of the effect of splitting BuMappingLock. Two backend processes can simultaneously hold respective BuMappingLock partitions in exclusive mode to insert new data entries. If BuMappingLock 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 BuMappingLock 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.

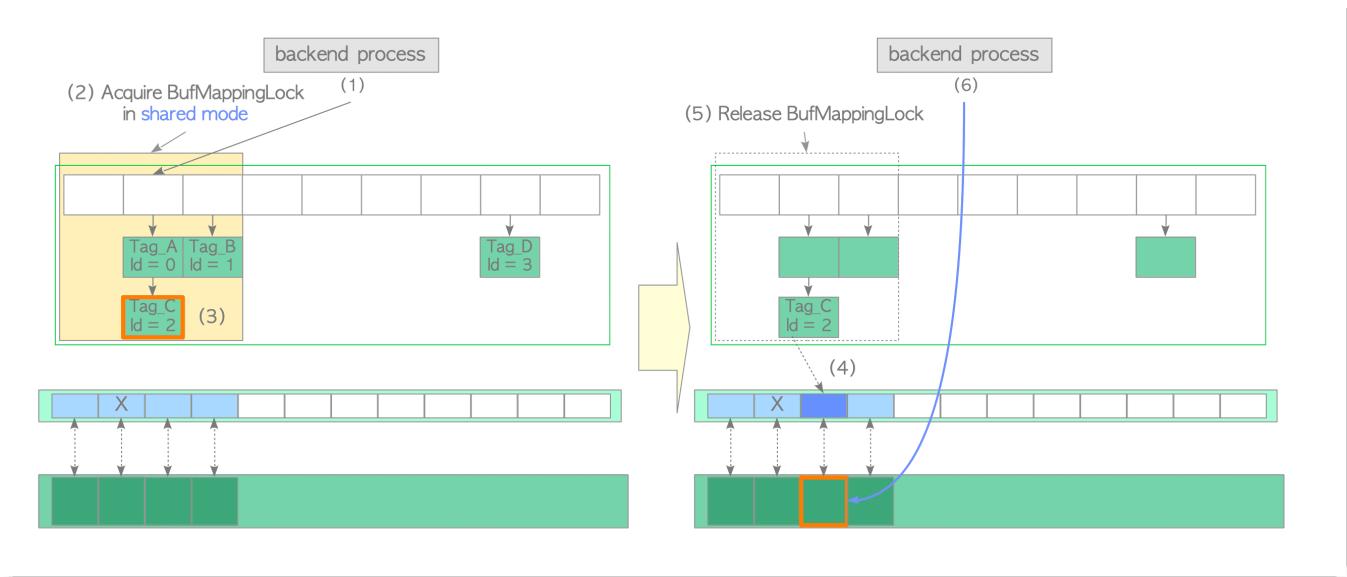
The behavior of the *ReadBufferExtended* function depends on three logical cases. Each case is described in the following subsections. In addition, the PostgreSQL *clock sweep* page replacement algorithm is described in the final subsection.

### 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.

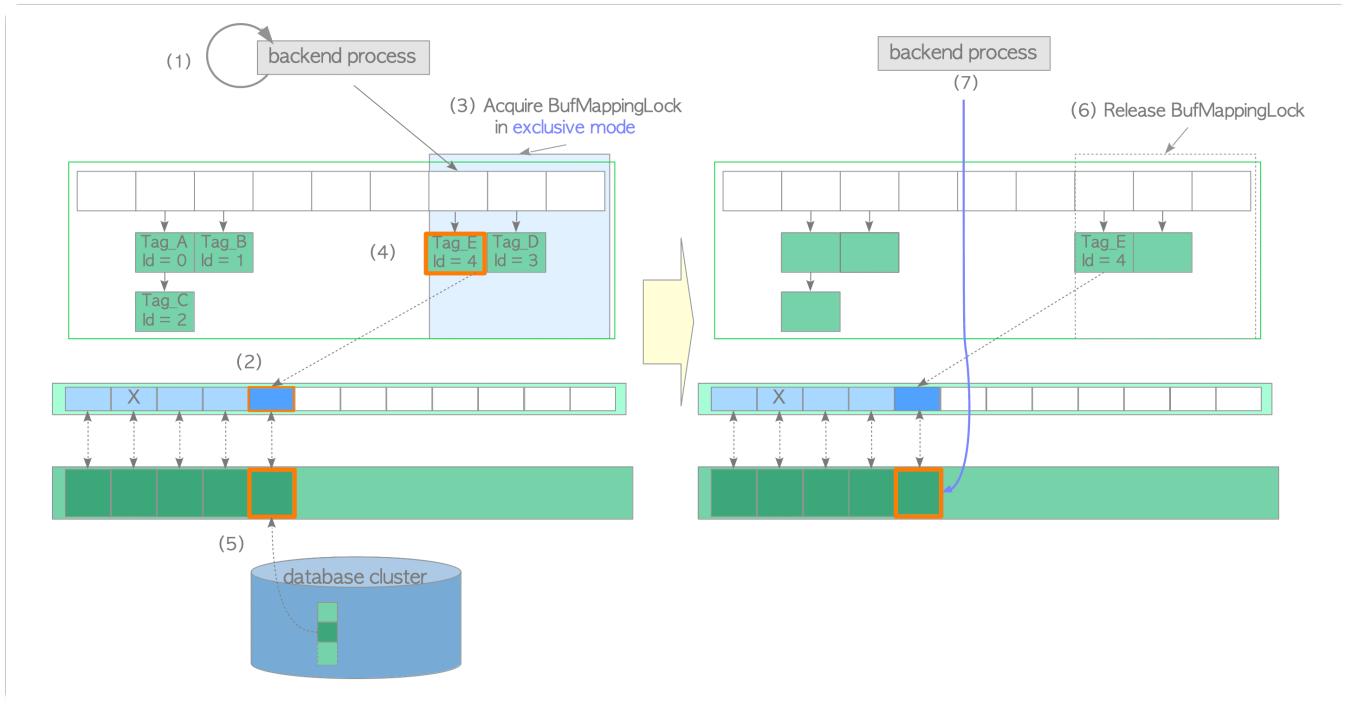
## 8.4.2. Loading a Page from Storage to Empty Slot

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

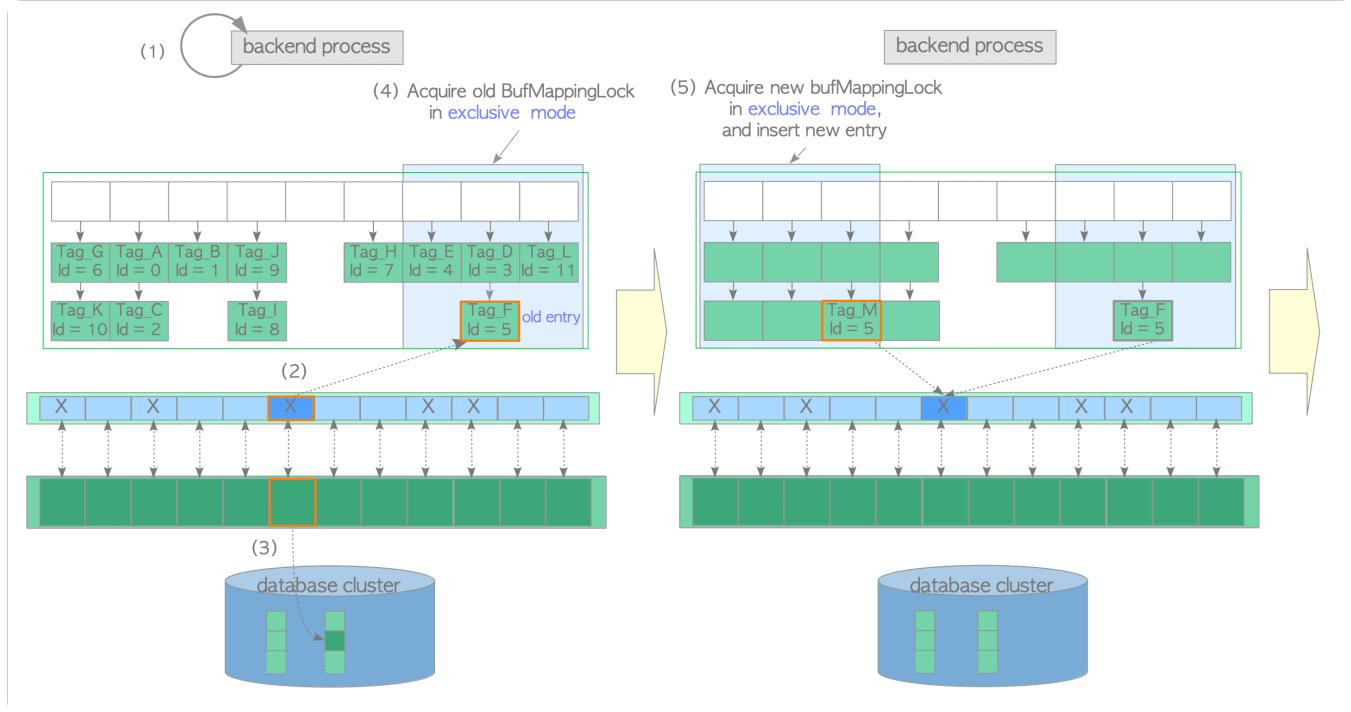
In this case, assume that all buffer pool slots are occupied by pages but the desired page is not stored. The buffer manager performs the following steps:

- (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.**



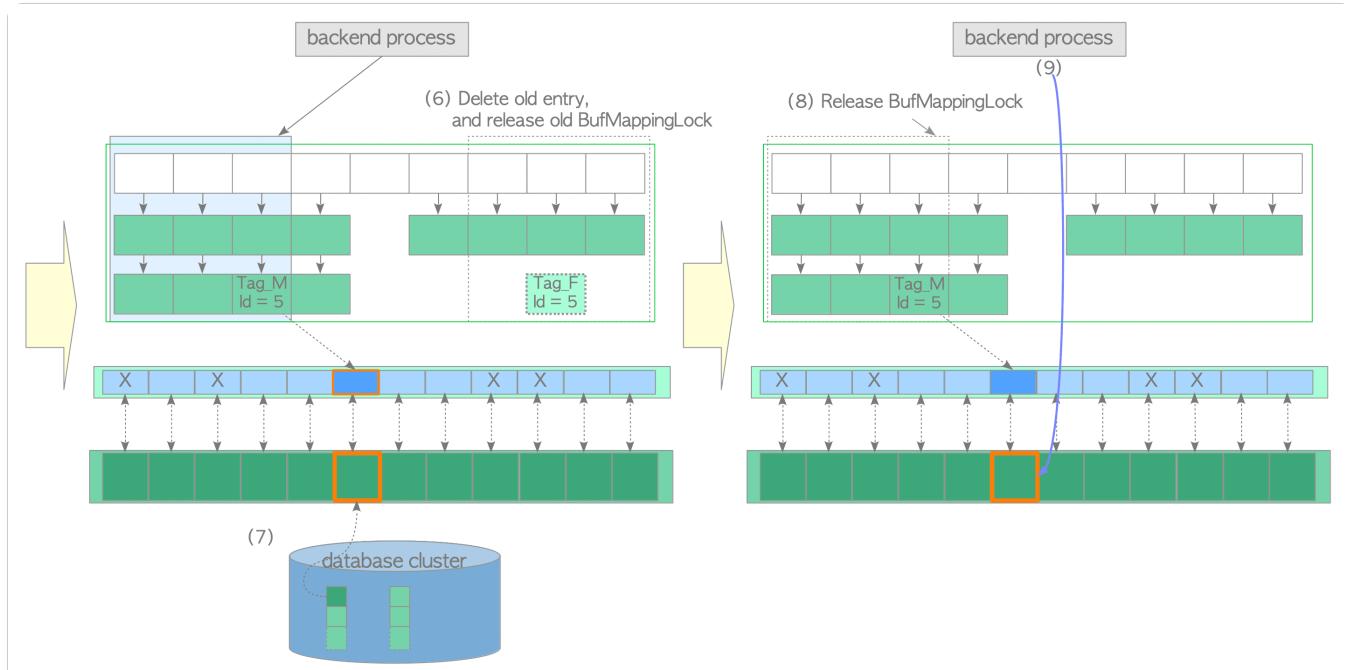
(6) Delete the old entry from the buffer table, and release the old BufMappingLock partition.

(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**

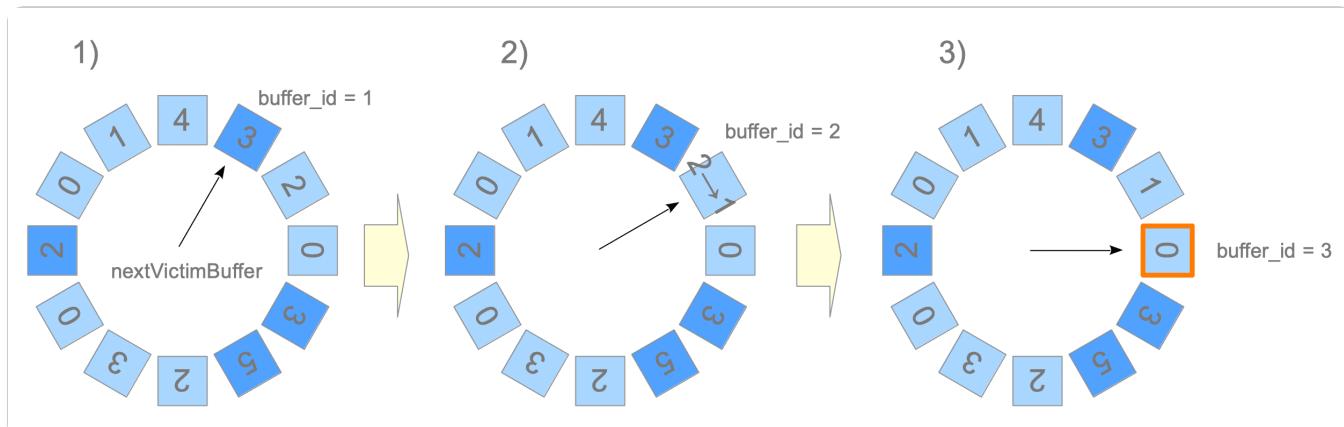
- (1) **Obtain** the candidate *buffer descriptor* pointed by the nextVictimBuffer
- (2) **IF** the candidate descriptor is *unpinned* **THEN**
- (3)   **IF** the candidate descriptor's *usage\_count == 0* **THEN**  
        **BREAK WHILE LOOP** /\* the corresponding slot of this descriptor is victim slot. \*/
- ELSE**  
        **Decrease** the candidate 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 *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 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** 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.

### ❶ 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.

## ❶ 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)

# Chapter 9

## Write Ahead Logging — WAL

---

**T**ransaction logs are an essential part of databases because they ensure that no data is lost even when a system failure occurs. They are a history log of all changes and actions in a database system. This ensures that no data is lost due to failures, such as a power failure or a server crash. The log contains sufficient information about each transaction that has already been executed, so the database server can recover the database cluster by replaying the changes and actions in the transaction log in the event of a server crash.

In the field of computer science, **WAL** is an acronym for **Write-Ahead Logging**, which is a protocol or rule that requires both changes and actions to be written to a transaction log. However, in PostgreSQL, WAL is also an acronym for **Write Ahead Log**. In PostgreSQL, the term WAL is used interchangeably with transaction log, and it also refers to the implemented mechanism for writing actions to a transaction log (WAL). Although this can be confusing, this document will adopt the PostgreSQL definition.

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 the WAL mechanism is essential for system integrations and administration using PostgreSQL, the complexity of this mechanism makes it impossible to summarize its description in brief. Therefore, the complete explanation of WAL in PostgreSQL is as follows:

- 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

### 9.1. Overview

Let's take a look at the overview of the WAL mechanism. To clarify the issue that WAL solves, the first subsection shows what happens when a crash occurs if PostgreSQL does not implement WAL. The second subsection introduces some key concepts and shows an overview of the main subjects in this chapter: the writing of WAL data and the database recovery process. The final subsection completes the overview of WAL by adding one more key concept.

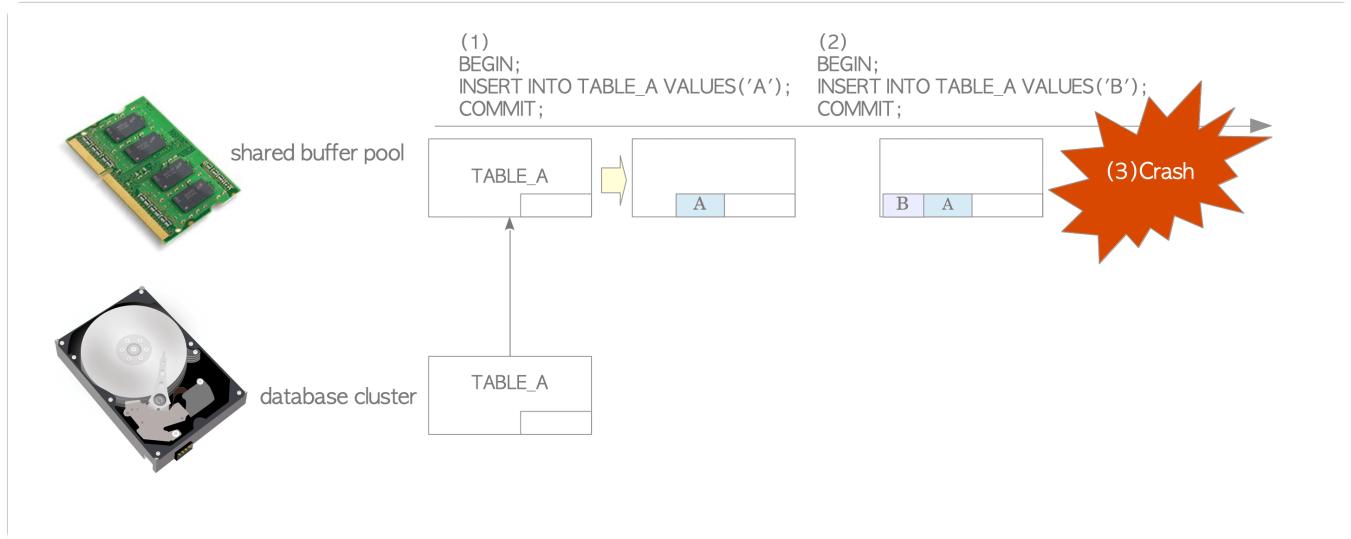
In this section, to simplify the description, the table TABLE\_A which contains just one page has been used.

### 9.1.1. Insertion Operations without WAL

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

Assume that we insert some data tuples into TABLE\_A on PostgreSQL which does not implement the WAL feature. This situation is illustrated in Fig. 9.1.

**Fig. 9.1. Insertion operations without WAL.**



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

Therefore, a database without WAL is vulnerable to system failures.

#### ❶ Historical Info

Before WAL was introduced (versions 7.0 or earlier), PostgreSQL did synchronous writes to the disk by issuing a sync system call whenever a page was changed in memory in order to ensure durability. This made modification commands such as INSERT and UPDATE very poor-performing.

### 9.1.2. Insertion Operations and Database Recovery

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

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 or aborts. (To be precise, the writing of XLOG records may occur in other cases. The details will be described in Section 9.5.) The **LSN (Log Sequence Number)** of an XLOG record represents the location where its record is written on the transaction log. The LSN of a record is used as the unique id of the XLOG record.

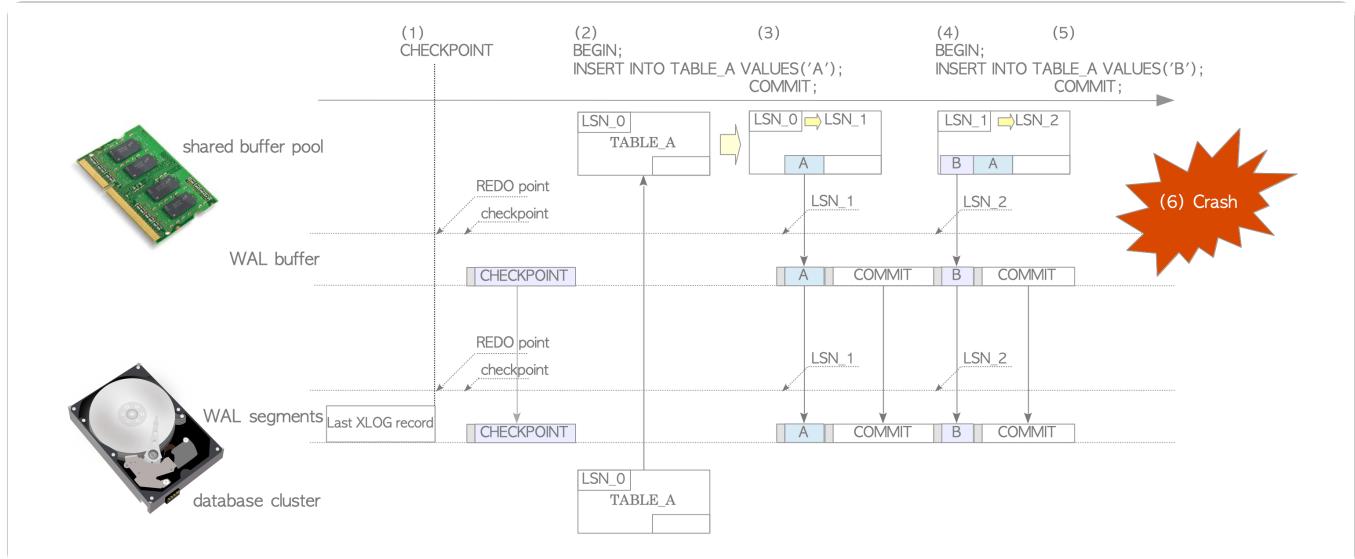
When considering how a database system recovers, one question that may arise is: what point does PostgreSQL start to recover from? The answer is the **REDO point**. That is the location to write the XLOG record at the moment when the latest **checkpoint** is started. (Checkpoints in PostgreSQL are described in Section 9.7.) In fact, the database recovery process is closely linked to the *checkpoint process*, and both of these processes 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, the following is a description of the tuple insertion with WAL. See Fig. 9.2 and the following description. (Also refer to this slide.)

**Fig. 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) When we issue the first INSERT statement, PostgreSQL loads the TABLE\_A 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*.

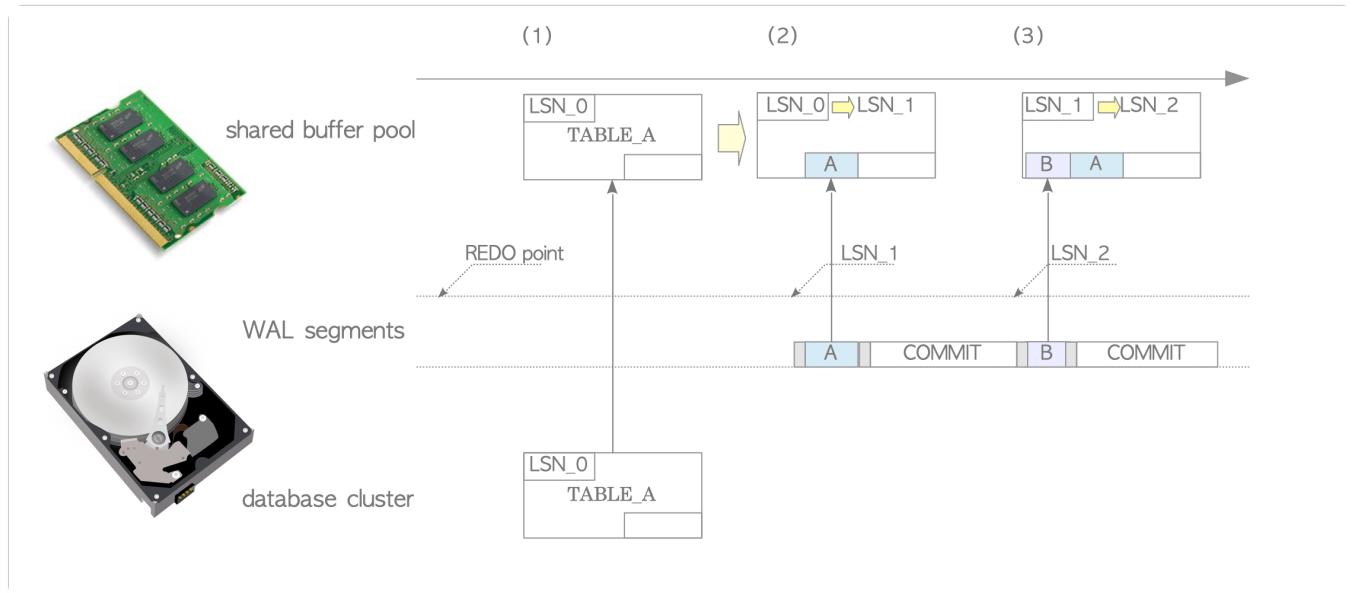
(4) When we issue the second INSERT statement, PostgreSQL inserts a new tuple into the page, creates and writes this tuple's XLOG record to the WAL buffer at *LSN\_2*, and updates the TABLE\_A's LSN from *LSN\_1* to *LSN\_2*.

(5) When this statement's transaction commits, PostgreSQL operates in the same manner as in step (3).

(6) Imagine that an operating system failure occurs. Even though all of the data in the shared buffer pool is lost, all modifications to the page have been written to 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 recovery-mode by restarting. See Fig. 9.3 (and this slide). PostgreSQL will sequentially read and replay XLOG records within the appropriate WAL segment files from the REDO point.

**Fig. 9.3. Database recovery using WAL.**



- (1) PostgreSQL reads the XLOG record of the first INSERT statement from the appropriate WAL segment file, and loads the TABLE\_A page from the database cluster into the shared buffer pool.
- (2) Before trying to replay the XLOG record, PostgreSQL compares the XLOG record's LSN with the corresponding page's LSN. the reason for doing this will be described in Section 9.8. The rules for replaying XLOG records are as follows:  
If the XLOG record's LSN is larger than the page's LSN, the data-portion of the XLOG record is inserted into the page, and the page's LSN is updated to the XLOG record's LSN.  
On the other hand, if the XLOG record's LSN is smaller, there is nothing to do other than to read next WAL record.

In this example, the XLOG record is replayed since the XLOG record's LSN (*LSN\_1*) is larger than the TABLE\_A's LSN (*LSN\_0*). Then, TABLE\_A's LSN is updated from *LSN\_0* to *LSN\_1*.

(3) 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 **REDO log**.



PostgreSQL does not support UNDO log.

Although writing XLOG records certainly costs a certain amount, it is nothing compared to writing the entire modified pages. We are confident that the benefit we gain, namely system failure tolerance, is greater than the amount we pay.

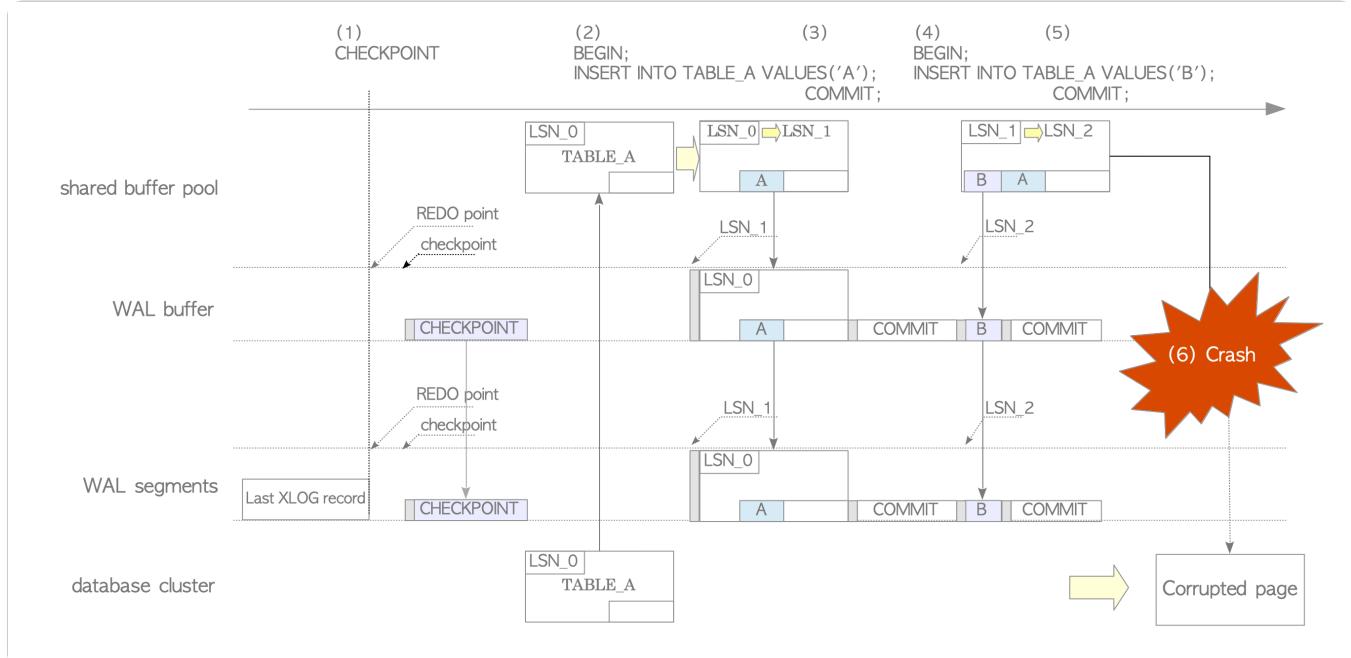
### 9.1.3. Full-Page Writes

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 called **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 an XLOG record during the first change of each page after every checkpoint. (This is the default setting.) In PostgreSQL, such a XLOG record containing the entire page is called a **backup block** (or **full-page image**).

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

**Fig. 9.4. Full page writes.**

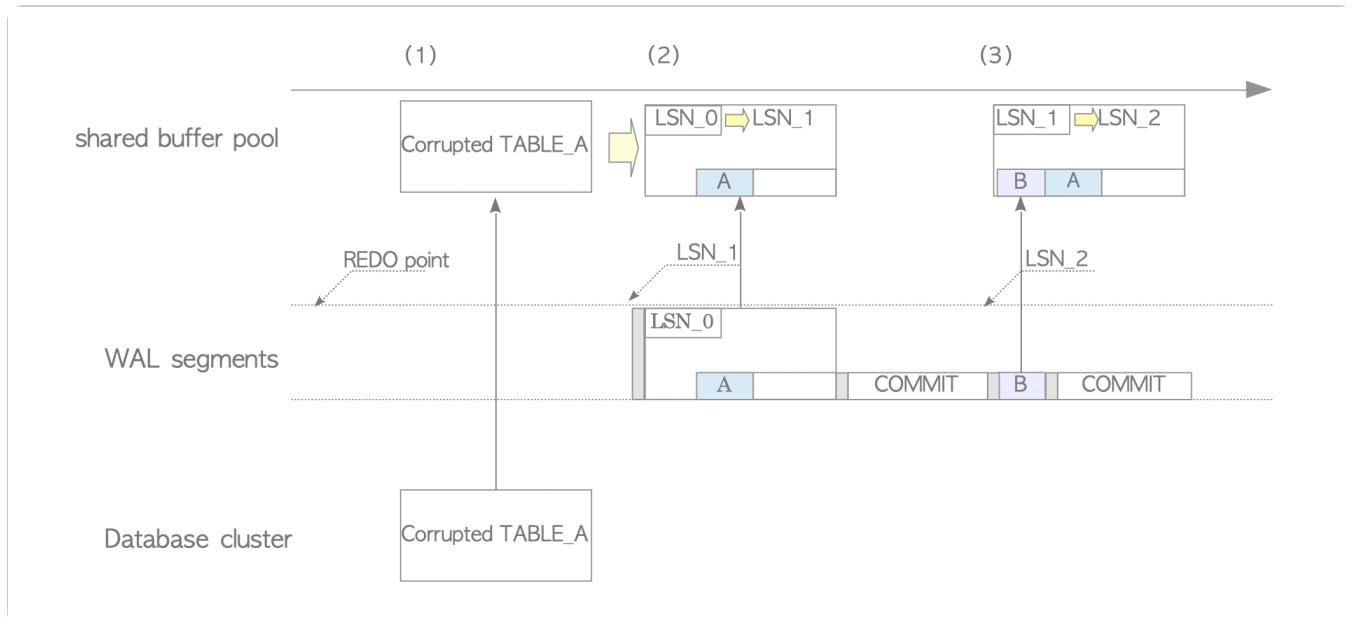


(1) The checkpointer starts a checkpoint process.

- (2) When we insert the first INSERT statement, PostgreSQL operates in the same way as in the previous subsection, except that this XLOG record is the *backup block* of this page, because this is the first writing of this page after the latest checkpoint. (In other words, it contains the entire page.)
- (3) As this transaction commits, PostgreSQL operates in the same way as in the previous subsection.
- (4) When we insert the second INSERT statement, PostgreSQL operates in the same way 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 way as in the previous subsection.
- (6) To demonstrate the effectiveness of full-page writes, let's consider the case in which the TABLE\_A page on the storage has been corrupted due to an operating system failure that occurred while the background writer was writing it to the storage (HDD or SSD).

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

**Fig. 9.5. Database recovery with backup block.**



- (1) PostgreSQL reads the XLOG record of the first INSERT statement and loads the corrupted TABLE\_A 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 way as the instruction in the previous subsection.

In this way, PostgreSQL can recover the database even if some data write errors occur due to a process or operating system crash.

## ❶ WAL, Backup, and Replication

As mentioned above, WAL can prevent data loss due to process or operating system crashes. However, if a file system or media failure occurs, the data will be lost. To deal with such failures, PostgreSQL provides online backup and replication features.

If online backups are taken regularly, the database can be restored from the most recent backup, even if a media failure occurs. However, it is important to note that the changes made after taking the last backup cannot be restored.

The synchronous replication feature can store all changes to another storage or host in real time. This means that if a media failure occurs on the primary server, the data can be restored from the secondary server.

For more information, see chapters 10 and 11, respectively.

## 9.2. Transaction Log and WAL Segment Files

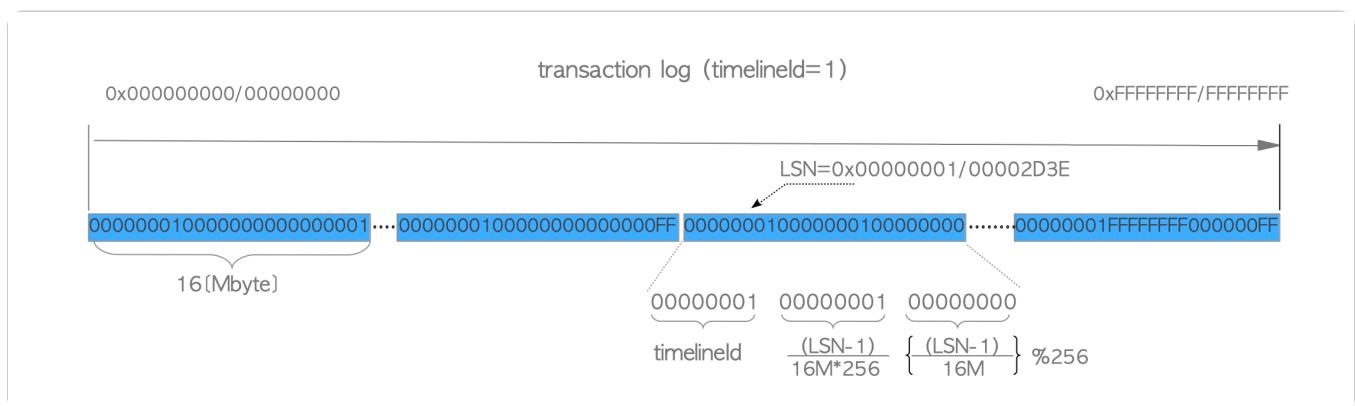
Logically, PostgreSQL writes XLOG records into a virtual file that is 8 bytes long (16 exabytes).

Since a transaction log capacity is effectively unlimited and so can be said that 8-bytes of address space is vast enough, it is impossible for us to handle a file with a capacity of 8 bytes. Therefore, a transaction log in PostgreSQL is divided into files of 16 megabytes, by default, each of which is known as a *WAL segment*. See Fig. 9.6.

### ❶ WAL segment file size

In versions 11 or later, the size of WAL segment file can be configured using `--wal-segsize` option when PostgreSQL cluster is created by `initdb` command.

**Fig. 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:

$$\text{WAL segment file name} = \text{timelineld} + (\text{uint32}) \frac{\text{LSN} - 1}{16M * 256} + (\text{uint32}) \left( \frac{\text{LSN} - 1}{16M} \right) \% 256$$

### ❶ timelineld

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 0000000100000000000000000001. If the first one has been filled up with the writing of XLOG records, the second one 0000000100000000000000000002 would be provided. Files are used in ascending order in succession. After 00000001000000000000FF has been filled up, the 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 0000000100000001000000FF has been filled up, 000000010000000200000000 will be provided, and so on.

### 🎓 pg\_xlogfile\_name / pg\_walfile\_name

Using the built-in function *pg\_xlogfile\_name* (versions 9.6 or earlier) or *pg\_walfile\_name* (versoin 10 or later), we can find the WAL segment file name that contains the specified LSN. An example is shown below:

```
testdb=# SELECT pg_xlogfile_name('1/00002D3E'); # In versions 10 or later, "SELECT pg_walfile_name('1/00002D3E');"
          pg_xlogfile_name
-----
000000010000000100000000
(1 row)
```

## 9.3. Internal Layout of WAL Segment

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

```
typedef XLogPageHeaderData *XLogPageHeader;

/*
 * When the XLP_LONG_HEADER flag is set, we store additional fields in the
 * page header. (This is ordinarily done just in the first page of an
 * XLOG file.) The additional fields serve to identify the file accurately.
 */
typedef struct XLogLongPageHeaderData
{
    XLogPageHeaderData std;           /* standard header fields */
    uint64        xlp_sysid;         /* system identifier from pg_control */
}
uint32        xlp_seg_size;      /* just as a cross-check */
uint32        xlp_xlog_blkksz;   /* just as a cross-check */
} XLogLongPageHeaderData;
/*
 * Each page of XLOG file has a header like this:
 */
#define XLOG_PAGE_MAGIC 0xD113 /* can be used as WAL version indicator */
```

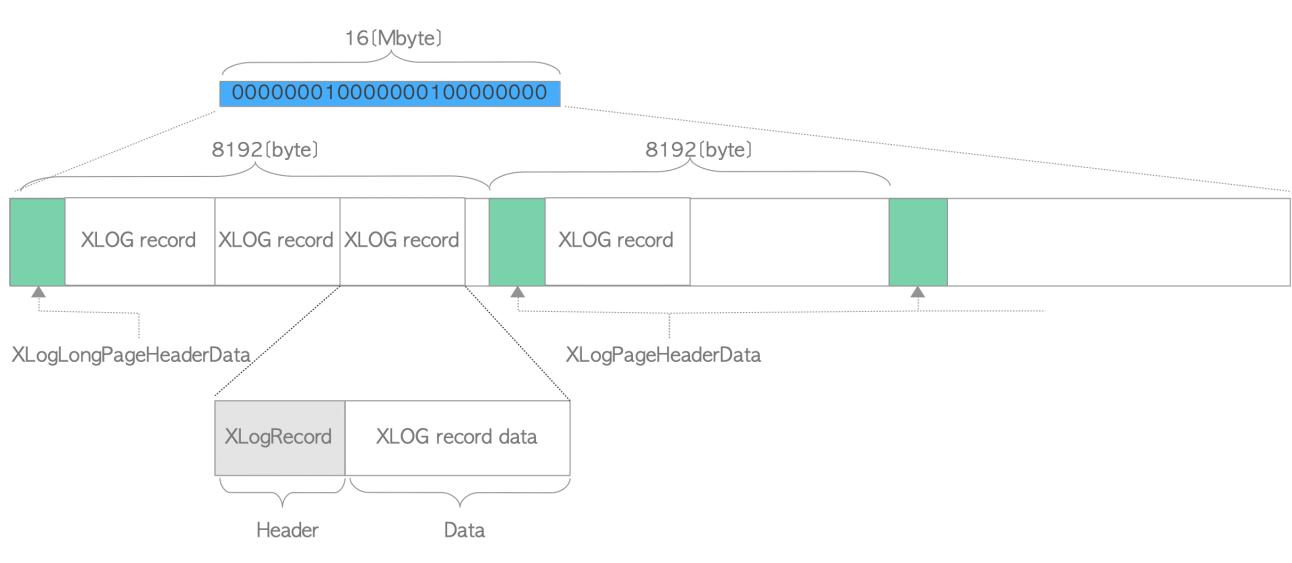
```

typedef struct XLogPageHeaderData
{
    uint16          xlp_magic;           /* magic value for correctness checks
*/
    uint16          xlp_info;            /* flag bits, see below */
    TimeLineID     xlp_tli;             /* TimeLineID of first record on page
*/
    XLogRecPtr     xlp_pageaddr;        /* XLOG address of this page */

/*
 * When there is not enough space on current page for whole record, we
 * continue on the next page. xlp_rem_len is the number of bytes
 * remaining from a previous page; it tracks xl_tot_len in the initial
 * header. Note that the continuation data isn't necessarily aligned.
 */
    uint32          xlp_rem_len;         /* total len of remaining data for record */
} XLogPageHeaderData;

```

**Fig. 9.7. Internal layout of a WAL segment file.**



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

## 9.4. Internal Layout of XLOG Record

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

### 9.4.1. Header Portion of XLOG Record

All XLOG records have a general header portion defined by the XLogRecord structure. Here, the structure of 9.4 and earlier versions is shown below, although it has been 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. This variable was removed in
ver. 9.5. */
    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;

```

## ❶ The Header Portion of XLOG Record in versions 9.5 or later.

In versions 9.5 or later, one variable (`xl_len`) has been removed the structure `XLogRecord` to refine the XLOG record format, which reduced the size by a few bytes.

```

typedef struct XLogRecord
{
    uint32          xl_tot_len;   /* total len of entire record */
    TransactionId  xl_xid;      /* xact id */
    XLogRecPtr     xl_prev;     /* ptr to previous record in log */
    uint8           xl_info;     /* flag bits, see below */
    RmgrId         xl_rmid;    /* resource manager for this record */
    /* 2 bytes of padding here, initialize to zero */
    pg_crc32c      xl_crc;      /* CRC for this record */
    /* XLogRecordBlockHeaders and XLogRecordDataHeader follow, no padding */
} XLogRecord;

```

Apart from two variables, most of the variables are so obvious that they do not need to be described.

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:

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 of how resource managers work:

- If an `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()` is selected according to the `xl_info` and replays this XLOG record.

- Similarly, for an 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.
- When a transaction commits, the header variables *xl\_rmid* and *xl\_info* of its XLOG record are set to 'RM\_XACT' and 'XLOG\_XACT\_COMMIT', respectively. When recovering the database cluster, the function *xact\_redo\_commit()* replays this record.



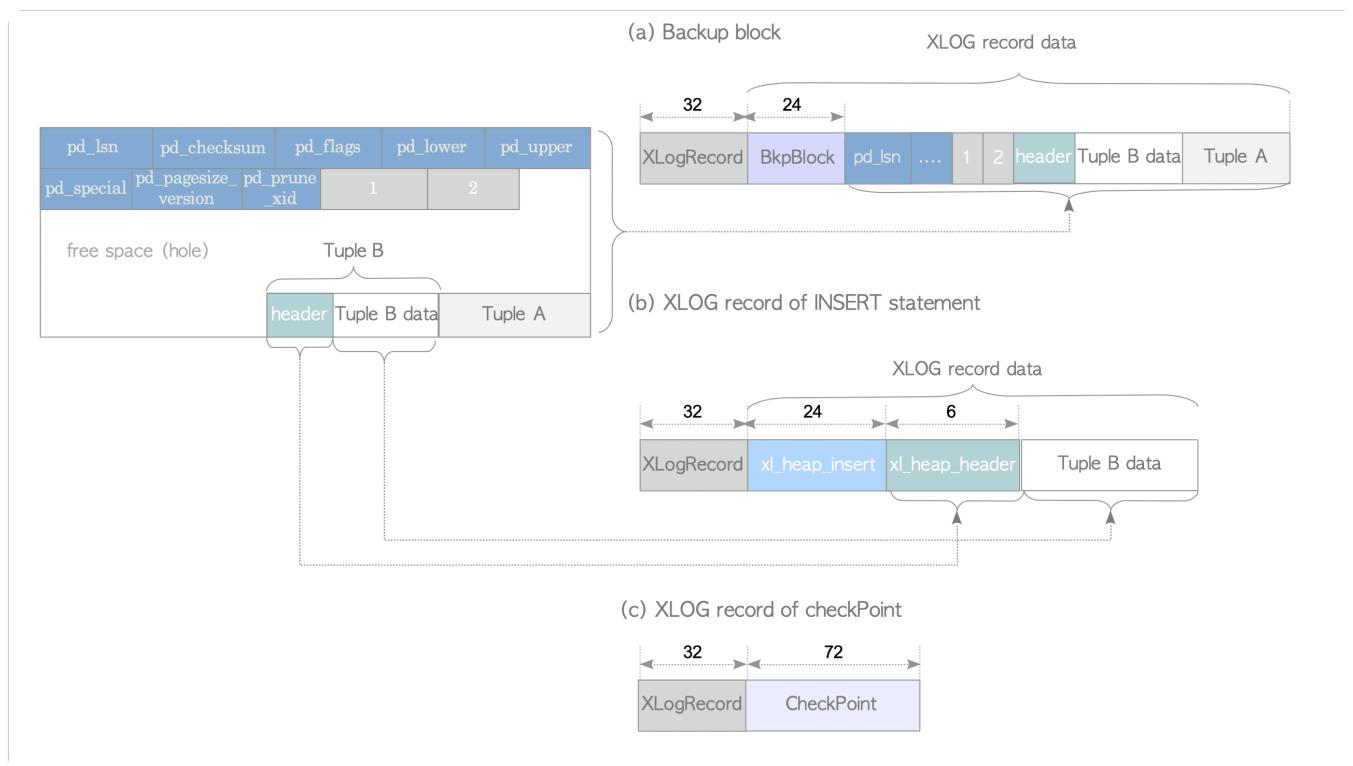
XLogRecord structure in versions 9.4 or earlier is defined in `src/include/access/xlog.h` and that of versions 9.5 or later is defined in `src/include/access/xlogrecord.h`.

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 (versions 9.4 or earlier)

The data portion of an XLOG record can be classified into either a backup block (which contains the entire page) or a non-backup block (which contains different data depending on the operation).

**Fig. 9.8. Examples of XLOG records (versions 9.4 or earlier).**



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

### 9.4.2.1. Backup Block

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

1. The XLogRecord structure (header portion)
2. The BkpBlock structure

### 3. The entire page, except for its free space

The `BkpBlock` structure contains the variables that identify the page in the database cluster (i.e., the *relfilenode* and *fork number* of the relation that contains the page, and the page's *block number*), as well as the starting position and length of the page's free space.

```
typedef struct BkpBlock @ include/access/xlog_internal.h
{
    RelFileNode node;          /* relation containing block */
    ForkNumber fork;           /* fork within the relation */
    BlockNumber block;         /* block number */
    uint16      hole_offset;   /* number of bytes before "hole" */
    uint16      hole_length;   /* number of bytes in "hole" */

    /* ACTUAL BLOCK DATA FOLLOWS AT END OF STRUCT */
} BkpBlock;
```

#### 9.4.2.2. Non-Backup Block

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

1. The `XLogRecord` (header-portion) structure
2. the `xl_heap_insert` structure
3. The inserted tuple, with a few bytes removed

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

One more example will be shown here. See Fig. 9.8(c). The XLOG record for a checkpoint record is quite simple; it is composed of two data structures:

1. the `XLogRecord` structure (header-portion)
2. the `Checkpoint` structure, which contains its checkpoint information (see more detail in Section 9.7)



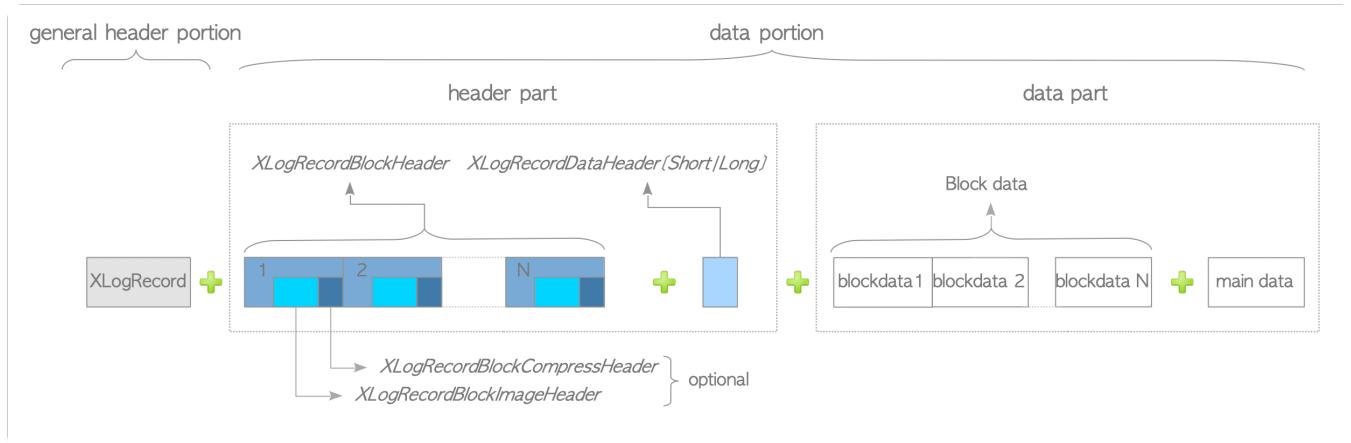
The xl\_heap\_header structure is defined in src/include/access/htup.h while the CheckPoint structure is defined in src/include/catalog/pg\_control.h.

### 9.4.3. Data Portion of XLOG Record (versions 9.5 or later)

In versions 9.4 or earlier, there was no common format for XLOG records, so each resource manager had to define its own format. This made it increasingly difficult to maintain the source code and implement new features related to WAL. To address this issue, a common structured format that is independent of resource managers was introduced in version 9.5.

The data portion of an XLOG record can be divided into two parts: header and data. See Fig. 9.9.

**Fig. 9.9. Common XLOG record format.**



The header part contains zero or more XLogRecordBlockHeaders and zero or one XLogRecordDataHeaderShort (or XLogRecordDataHeaderLong). It must contain at least one of these. When the record stores a full-page image (i.e., a backup block), the XLogRecordBlockHeader includes the XLogRecordBlockImageHeader, and also includes the XLogRecordBlockCompressHeader if its block is compressed.

```
/*
 * Header info for block data appended to an XLOG record.
 *
 * 'data_length' is the length of the rmgr-specific payload data associated
 * with this block. It does not include the possible full page image, nor
 * XLogRecordBlockHeader struct itself.
 *
 * Note that we don't attempt to align the XLogRecordBlockHeader struct!
 * So, the struct must be copied to aligned local storage before use.
 */
typedef struct XLogRecordBlockHeader
{
    uint8          id;           /* block reference ID */
    uint8          fork_flags;   /* fork within the relation, and flags */
    uint16         data_length;  /* number of payload bytes (not including page
                                * image) */

    /* If BKPBLOCK_HAS_IMAGE, an XLogRecordBlockImageHeader struct follows */
    /* If BKPBLOCK_SAME_REL is not set, a RelFileLocator follows */
    /* BlockNumber follows */
} XLogRecordBlockHeader;

/*
 * The fork number fits in the lower 4 bits in the fork_flags field. The upper
 * bits are used for flags.
 */
```

```

#define BKPBLOCK_FORK_MASK 0x0F
#define BKPBLOCK_FLAG_MASK 0xF0
#define BKPBLOCK_HAS_IMAGE 0x10 /* block data is an XLogRecordBlockImage */
#define BKPBLOCK_HAS_DATA 0x20
#define BKPBLOCK_WILL_INIT 0x40 /* redo will re-init the page */
#define BKPBLOCK_SAME_REL 0x80 /* RelFileLocator omitted, same as
                                * previous */

/*
 * Additional header information when a full-page image is included
 * (i.e. when BKPBLOCK_HAS_IMAGE is set).
 *
 * The XLOG code is aware that PG data pages usually contain an unused "hole"
 * in the middle, which contains only zero bytes. Since we know that the
 * "hole" is all zeros, we remove it from the stored data (and it's not counted
 * in the XLOG record's CRC, either). Hence, the amount of block data actually
 * present is (BLCKSZ - ). 
 *
 * Additionally, when wal_compression is enabled, we will try to compress full
 * page images using one of the supported algorithms, after removing the
 * "hole". This can reduce the WAL volume, but at some extra cost of CPU spent
 * on the compression during WAL logging. In this case, since the "hole"
 * length cannot be calculated by subtracting the number of page image bytes
 * from BLCKSZ, basically it needs to be stored as an extra information.
 * But when no "hole" exists, we can assume that the "hole" length is zero
 * and no such an extra information needs to be stored. Note that
 * the original version of page image is stored in WAL instead of the
 * compressed one if the number of bytes saved by compression is less than
 * the length of extra information. Hence, when a page image is successfully
 * compressed, the amount of block data actually present is less than
 * BLCKSZ - the length of "hole" bytes - the length of extra information.
 */
typedef struct XLogRecordBlockImageHeader
{
    uint16          length;           /* number of page image bytes */
    uint16          hole_offset;     /* number of bytes before "hole" */
    uint8           bimg_info;       /* flag bits, see below */

    /*
     * If BKPIIMAGE_HAS_HOLE and BKPIIMAGE_COMPRESSED(), an
     * XLogRecordBlockCompressHeader struct follows.
     */
} XLogRecordBlockImageHeader;

/* Information stored in bimg_info */
#define BKPIIMAGE_HAS_HOLE           0x01 /* page image has "hole" */
#define BKPIIMAGE_APPLY              0x02 /* page image should be restored
                                             * during replay */
#define BKPIIMAGE_COMPRESS_PGLZ      0x04
#define BKPIIMAGE_COMPRESS_LZ4       0x08
#define BKPIIMAGE_COMPRESS_ZSTD      0x10

#define BKPIIMAGE_COMPRESSED(info) \
    ((info & (BKPIIMAGE_COMPRESS_PGLZ | BKPIIMAGE_COMPRESS_LZ4 | \
               BKPIIMAGE_COMPRESS_ZSTD)) != 0)

/*
 * Extra header information used when page image has "hole" and
 * is compressed.
 */
typedef struct XLogRecordBlockCompressHeader

```

```

{
    uint16          hole_length; /* number of bytes in "hole" */
} XLogRecordBlockCompressHeader;

/*
 * XLogRecordDataHeaderShort/Long are used for the "main data" portion of
 * the record. If the length of the data is less than 256 bytes, the short
 * form is used, with a single byte to hold the length. Otherwise the long
 * form is used.
 *
 * (These structs are currently not used in the code, they are here just for
 * documentation purposes).
*/
typedef struct XLogRecordDataHeaderShort
{
    uint8           id;                      /* XLR_BLOCK_ID_DATA_SHORT */
    uint8           data_length;   /* number of payload bytes */
} XLogRecordDataHeaderShort;

#define SizeOfXLogRecordDataHeaderShort (sizeof(uint8) * 2)

typedef struct XLogRecordDataHeaderLong
{
    uint8           id;                      /* XLR_BLOCK_ID_DATA_LONG */
    /* followed by uint32 data_length, unaligned */
} XLogRecordDataHeaderLong;

#define SizeOfXLogRecordDataHeaderLong (sizeof(uint8) + sizeof(uint32))

```

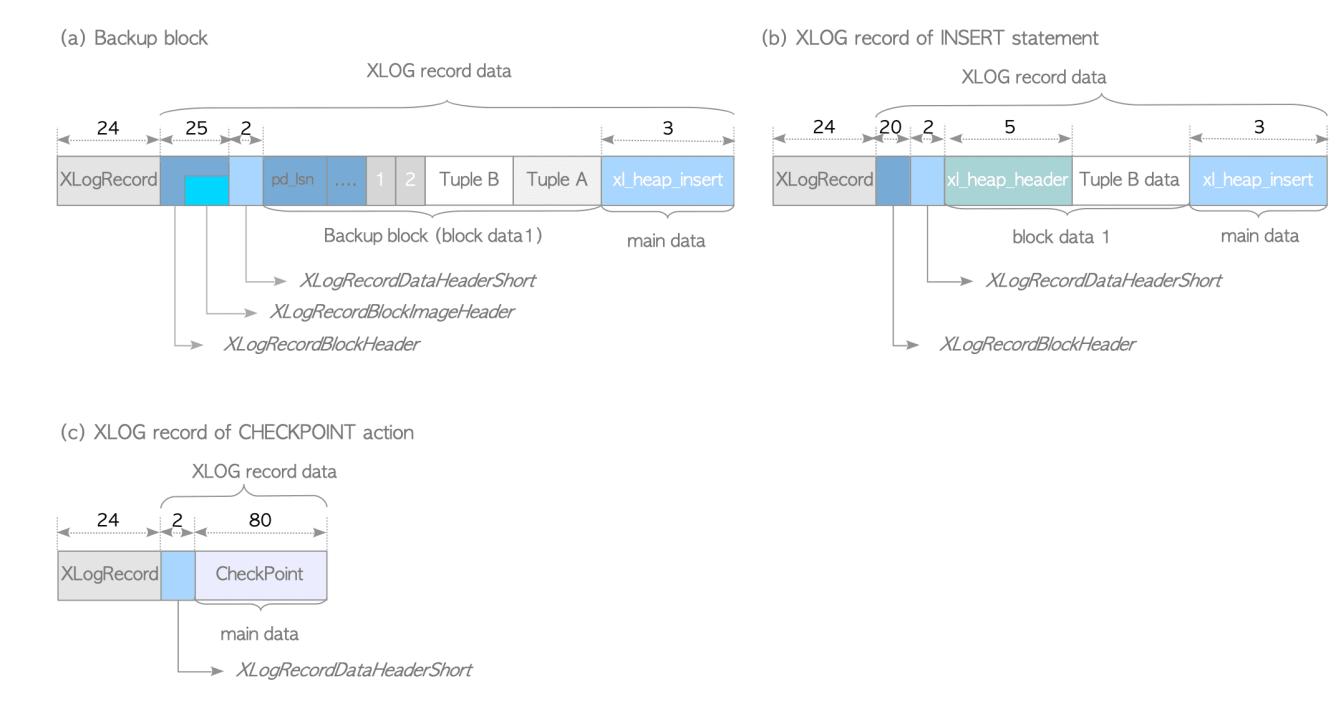
The data part is composed of zero or more block data and zero or one main data, which correspond to the XLogRecordBlockHeader(s) and to the XLogRecordDataHeader, respectively

## WAL compression

In versions 9.5 or later, full-page images within XLOG records can be compressed using the LZ compression method by setting the parameter wal\_compression = enable. In that case, the XLogRecordBlockCompressHeader structure will be added.

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

**Fig. 9.10. Examples of XLOG records (versions 9.5 or later).**



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

#### 9.4.3.1. Backup Block

The backup block created by an INSERT statement is shown in Fig. 9.10(a). It is composed of four data structures and one data object:

1. the XLogRecord structure (header-portions)
2. the XLogRecordBlockHeader structure, including one LogRecordBlockImageHeader structure
3. the XLogRecordDataHeaderShort structure
4. a backup block (block data)
5. the xl\_heap\_insert structure (main data)

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

The XLogRecordDataHeaderShort structure stores the length of the *xl\_heap\_insert* structure, which is the main data of the record. (See **i** below.)



The main data of an XLOG record that contains a full-page image is not used except in some special cases, such as logical decoding and speculative insertions. It is ignored when the record is replayed, making it redundant data. This may be improved in the future.

In addition, the main data of backup block records depends on the statements that create them. For example, an UPDATE statement appends *xl\_heap\_lock* or *xl\_heap\_updated*.

#### 9.4.3.2. Non-Backup Block

Next, I will describe the non-backup block record created by the INSERT statement (see Fig. 9.10(b)). It is composed of four data structures and one data object:

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

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

The new `xl_heap_insert` structure contains only two values: the *offset number* of this tuple within the block, and a *visibility flag*. It became very simple because the `XLogRecordBlockHeader` structure stores most of the data that was contained in the old `xl_heap_insert` structure.

As the final example, a checkpoint record is shown in the Fig. 9.10(c). It is composed of three data structures:

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



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`.

Although the new format is a little complicated for us, it is well-designed for the parsers of the resource managers. Additionally, the size of many types of XLOG records is usually smaller than the previous ones. The sizes of the main structures are shown in Figures 9.8 and 9.10, so you can calculate the sizes of those records and compare them. (The size of the new checkpoint is greater than the previous one, but it contains more variables.)

## 9.5. Writing of XLOG Records

Having finished the warm-up exercises, we are now ready to understand the writing of XLOG records. I will explain it as precisely as possible in this section.

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 pseudocode 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.
```

```

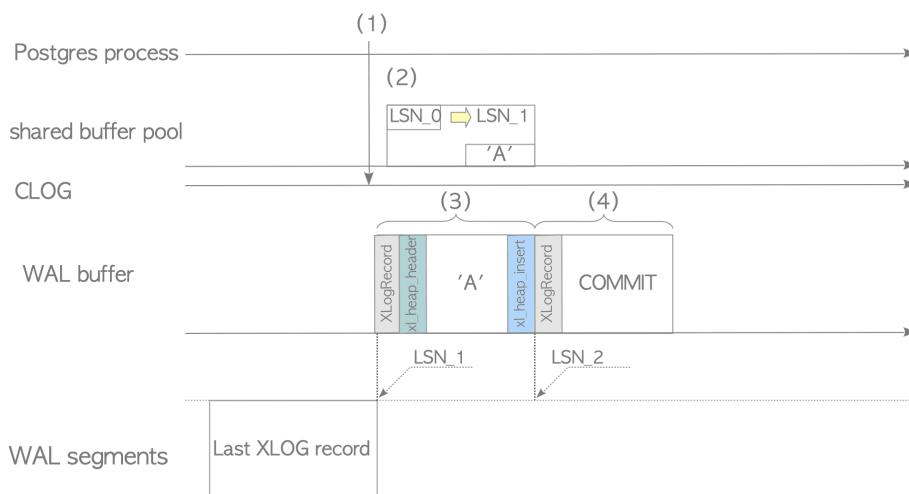
        */
(2) heap_insert() @heapam.c      /* Insert a tuple, creates a XLOG record,
                                * and invoke the function XLogInsert.
                                */
(3) XLogInsert() @xloginsert.c (9.4 or earlier, xlog.c)
        /* Write the XLOG record of the inserted tuple
         * to the WAL buffer, and update page's pd_lsn.
n.
        */
(4) finish_xact_command() @postgres.c    /* Invoke commit action.*/
    XLogInsert() @xloginsert.c (9.4 or earlier, xlog.c)
        /* Write a XLOG record of this commit action
         * to the WAL buffer.
        */
(5) XLogWrite() @xloginsert.c (9.4 or earlier, 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 pseudocode will be explained to help you understand the writing of XLOG records. See also Figs. 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 in the shared buffer pool, creates the XLOG record for that page, 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 is invoked to commit this transaction, creates the XLOG record for the commit action, and then the function *XLogInsert()* writes this record to the WAL buffer at *LSN\_2*.

**Fig. 9.11. Write-sequence of XLOG records.**



The format of these XLOG records is version 9.4.

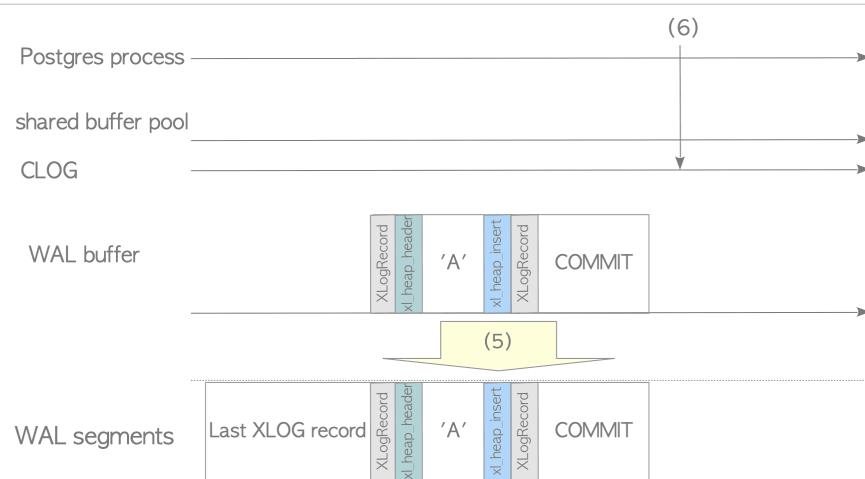
(5) The function *XLogWrite()* writes and flushes all XLOG records on the WAL buffer to the WAL segment file.

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.

**Fig. 9.12. Write-sequence of XLOG records (continued from Fig. 9.11).**



In the above example, the commit action caused the writing of XLOG records to the WAL segment, but such writing may be caused by any of the following:

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

If any of the 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 the committed transaction. Another example is a checkpoint action, which writes a XLOG record that contains general information about the checkpoint. Furthermore, the SELECT statement creates XLOG records in special cases, although 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, the XLOG records of modified pages are written to the WAL buffer.

PostgreSQL versions 15 and earlier do not support direct I/O, although it has been discussed. Refer 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.

## 9.6. WAL Writer Process

The WAL writer is a background process that periodically checks the WAL buffer and writes all unwritten XLOG records to the WAL segments. This process helps to avoid bursts of XLOG record writing. If the WAL writer is not enabled, writing XLOG records could be bottlenecked when a large amount of data is committed at once.

The WAL writer is enabled by default and cannot be disabled. The check interval is set to the configuration parameter `wal_writer_delay`, which defaults to 200 milliseconds.

## 9.7. Checkpoint Processing in PostgreSQL

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

1. The interval time set for `checkpoint_timeout` from the previous checkpoint has been elapsed (the default interval is 300 seconds (5 minutes)).
2. In versions 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 versions 9.5 or later, the total size of the WAL segment files in the `pg_wal` (in versions 9.6 or earlier, `pg_xlog`) directory has exceeded the value of the parameter `max_wal_size` (the default value is 1GB (64 files)).
4. The PostgreSQL server stops in `smart` or `fast` mode.
5. A superuser issues the `CHECKPOINT` command manually.



In versions 9.1 or earlier, as mentioned in **i** in Section 8.6, the background writer process did both checkpointing and dirty-page writing.

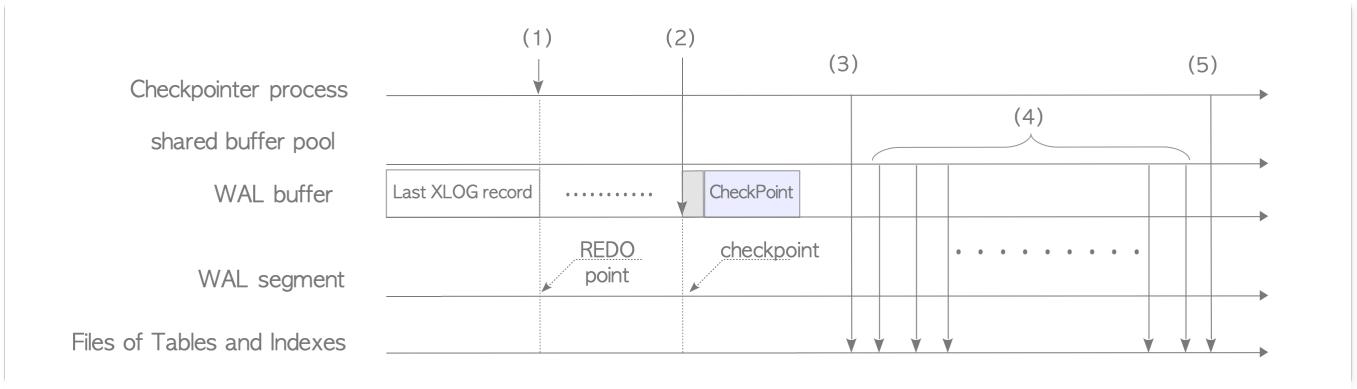
In the following subsections, the outline of checkpointing and the `pg_control` file, which holds the metadata of the current checkpoint, are described.

v

### 9.7.1. Outline of the Checkpoint Processing

The checkpoint process has two aspects: preparing for database recovery, and cleaning dirty pages in the shared buffer pool. In this subsection, we will focus on the former aspect and describe its internal processing. See Fig. 9.13 for an overview.

**Fig. 9.13. Internal processing of PostgreSQL Checkpoint.**



- (1) When a checkpoint process starts, the REDO point is stored in memory. The REDO point is the location of the XLOG record that was written at the moment the latest checkpoint was started. It is the starting point for database recovery.
- (2) A XLOG record of this checkpoint (i.e., the checkpoint record) is written to the WAL buffer. The data-portion of the record is defined by the CheckPoint structure, which contains several variables such as the REDO point stored in step (1).
- The location where the checkpoint record is written is also called the *checkpoint*.

```

typedef struct CheckPoint
{
    XLogRecPtr      redo;           /* next RecPtr available when we began to
                                    * create CheckPoint (i.e. REDO start point) */
    TimeLineID      ThisTimeLineID; /* current TLI */
    TimeLineID      PrevTimeLineID; /* previous TLI, if this record begins a new
                                    * timeline (equals ThisTimeLineID otherwise)
*/
    bool            fullPageWrites; /* current full_page_writes */
    uint32          nextXidEpoch;   /* higher-order bits of nextXid */
    TransactionId   nextXid;       /* next free XID */
    Oid             nextOid;        /* next free OID */
    MultiXactId    nextMulti;     /* next free MultiXactId */
    MultiXactOffset nextMultiOffset; /* next free MultiXact offset */
    TransactionId   oldestXid;     /* cluster-wide minimum datfrozenxid */
    Oid             oldestXidDB;   /* database with minimum datfrozenxid */
    MultiXactId    oldestMulti;   /* cluster-wide minimum datminmxid */
    Oid             oldestMultiDB; /* database with minimum datminmxid */
    pg_time_t       time;          /* time stamp of checkpoint */

/*
 * Oldest XID still running. This is only needed to initialize hot standby
 * mode from an online checkpoint, so we only bother calculating this for
 * online checkpoints and only when wal_level is hot_standby. Otherwise
 * it's set to InvalidTransactionId.
*/
    TransactionId   oldestActiveXid;
} CheckPoint;

```

- (3) All data in shared memory (e.g., the contents of the clog, etc.) is flushed to the storage.
- (4) All dirty pages in the shared buffer pool are gradually written and flushed to the storage.
- (5) The *pg\_control* file is updated. This file contains fundamental information such as the location where the checkpoint record was written (a.k.a. the checkpoint location). We will discuss this file in more detail later.

To summarize the description above from the perspective of database recovery, checkpointing creates a checkpoint record that contains the REDO point, and stores the checkpoint location and

other information in the *pg\_control* file. This allows PostgreSQL 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 essential for database recovery. If it is corrupted or unreadable, the recovery process cannot start because it cannot obtain a starting point.

Even though *pg\_control\_file* stores over 40 items, three items that will be needed in the next section are shown below:

- **State** – The state of the database server at the time the latest checkpoint was started. There are seven states in total:
    - '*start up*' is the state that system is starting up.
    - '*shut down*' is the state that the system is going down normally by the shutdown command.
    - '*in production*' is the state that the system is running.and so on.
  - **Latest checkpoint location** – The LSN Location of the latest checkpoint record.
  - **Prior checkpoint location** – The LSN Location of the prior checkpoint record. Note that this is deprecated in version 11; the details are described in [? below](#).

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:          1300
Catalog version number:            202306141
Database system identifier:        7250496631638317596
Database cluster state:           in production
pg_control last modified:         Thu Oct 26 15:16:38 2023
Latest checkpoint location:       0/16AF0090
Latest checkpoint's REDO location: 0/16AF0090
Latest checkpoint's REDO WAL file: 00000001000000000000000000000016
```

 Removal of prior checkpoint in PostgreSQL 11

PostgreSQL 11 or later only stores the WAL segments that contain the latest checkpoint or newer. Older segment files, which contains the prior checkpoint, are not stored to reduce the disk space used for saving WAL segment files under the pg\_wal subdirectory. See [this thread](#) in details.

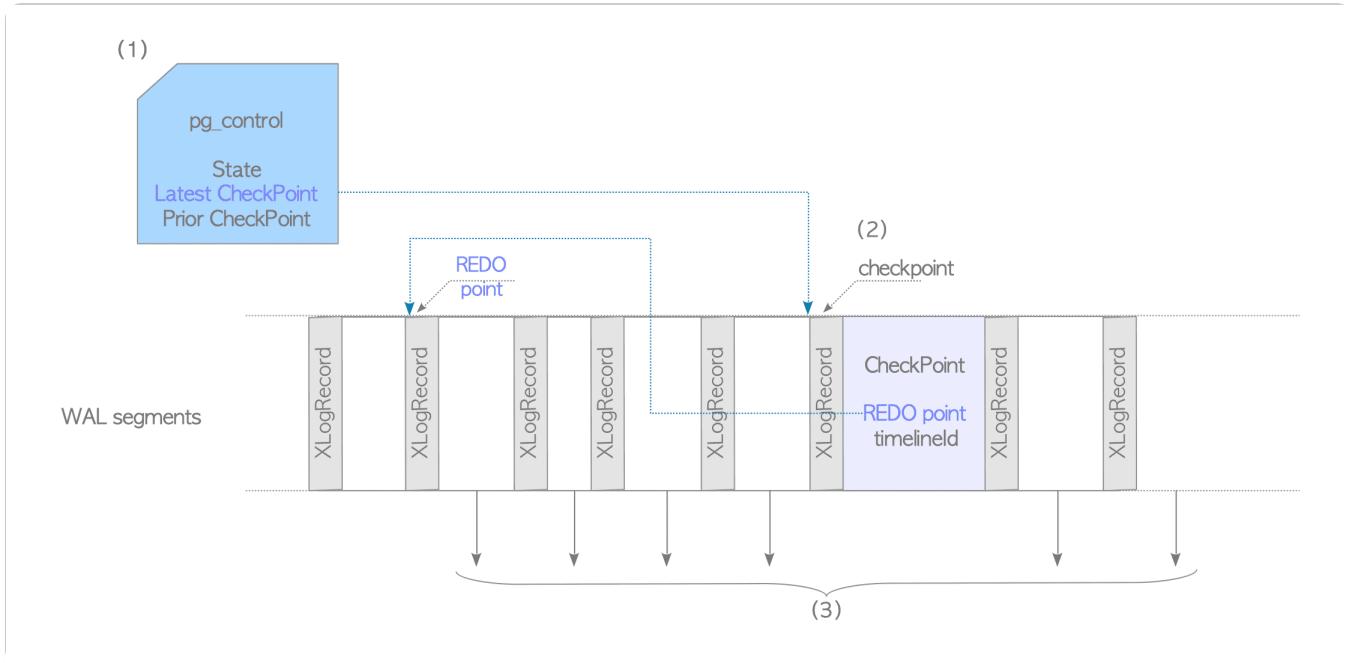
## 9.8. Database Recovery in PostgreSQL

PostgreSQL implements redo log-based recovery. If the database server crashes, PostgreSQL can restore the database cluster by sequentially replaying the XLOG records in the WAL segment files from the REDO point.

We have already talked about database recovery several times up to this section. Here, I will describe two things about recovery that have not been explained yet.

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

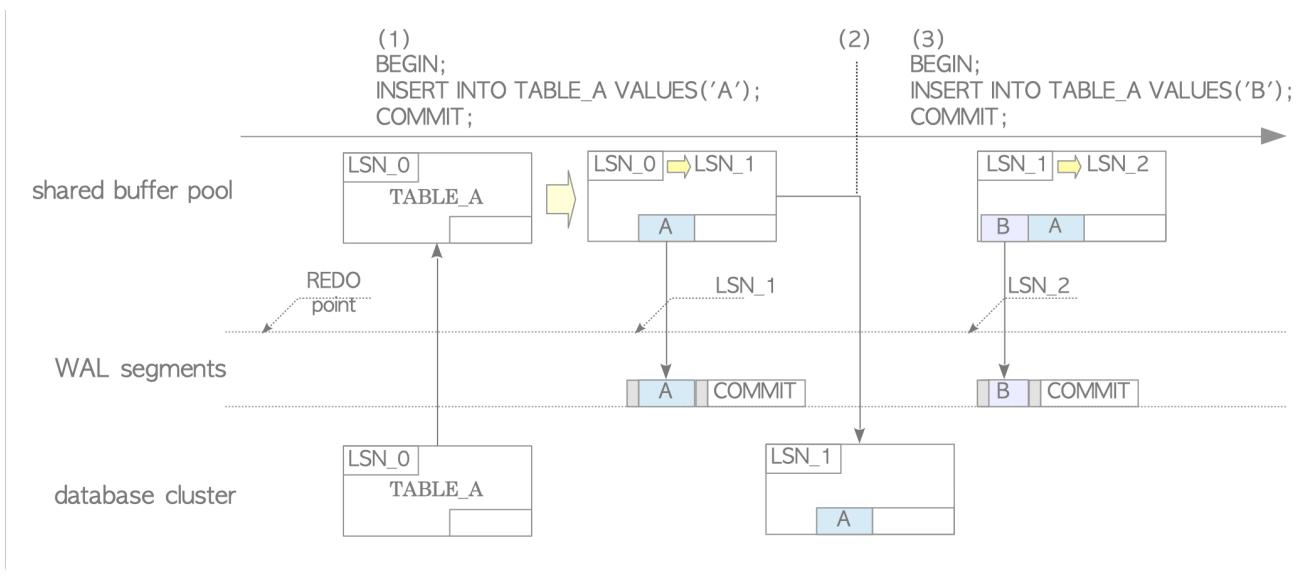
**Fig. 9.14. Details of the recovery process.**



- (1) PostgreSQL reads all items in the `pg_control` file when it starts. If the `state` item is '`in production`', PostgreSQL enters recovery-mode because this means that the database was not shut down normally. If it is '`shut down`', PostgreSQL enters normal startup-mode.
- (2) PostgreSQL reads the latest checkpoint record, the location of which is written in the `pg_control` file, from the appropriate WAL segment file. It then 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. (Note that the prior checkpoint is not stored in PostgreSQL 11 or later.)
- (3) The appropriate resource managers read and replay XLOG records in sequence from the REDO point until they reach the end of the latest WAL segment. When an XLOG record is replayed and if it is a backup block, it is overwritten on the corresponding table page regardless of its LSN. Otherwise, a (non-backup block) XLOG record is replayed only if the LSN of the record is greater than the `pd_lsn` of the 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, this will be explained using a specific example that emphasizes the need for this comparison. See Figs. 9.15 and 9.16. (Note that the WAL buffer is omitted to simplify the description.)

**Fig. 9.15. Insertion operations during the background writer working.**

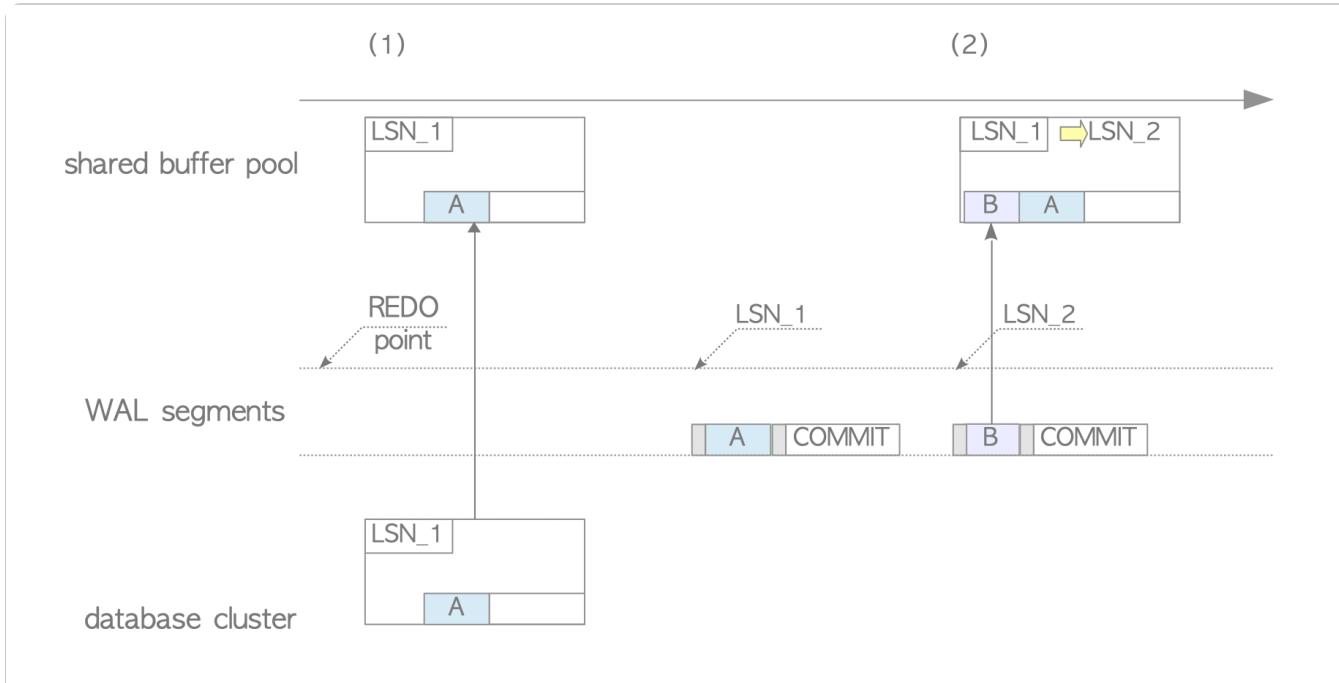


- (1) PostgreSQL inserts a tuple into the TABLE\_A, and writes an XLOG record at *LSN\_1*.
- (2) The background-writer process writes the TABLE\_A page to storage. At this point, this page's *pd\_lsn* is *LSN\_1*.
- (3) PostgreSQL inserts a new tuple into the TABLE\_A, and writes a XLOG record at *LSN\_2*. The modified page is not written into the storage yet.

Unlike the examples in overview, the TABLE\_A's page has been once written to the storage in this scenario.

Do shutdown with immediate-mode, and then start the database.

**Fig. 9.16. Database recovery.**



- (1) PostgreSQL loads the first XLOG record and the TABLE\_A page, but does not replay it because the LSN of the record (LSN\_1) is not larger than the LSN of the page (also LSN\_1). In fact, it is clear that there is no need to replay it.
- (2) Next, PostgreSQL replays the second XLOG record because the LSN of the record (LSN\_2) is greater than the current LSN of the TABLE\_A page (LSN\_1).

As you can see from this example, if the replay order of non-backup blocks is incorrect or non-backup blocks are replayed 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 replay order, non-backup block records should be replayed only if their 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 their LSN.

## 9.9. WAL Segment Files Management

PostgreSQL writes XLOG records to one of the WAL segment files stored in the pg\_wal subdirectory (in versions 9.6 or earlier, pg\_xlog subdirectory). A new WAL segment file is switched in if the old one has been filled up. The number of WAL files varies depending on several configuration parameters, as well as server activity. In addition, the management policy for WAL segment files has been improved in version 9.5.

The following subsections describe how WAL segment files are switched and managed.

### 9.9.1. WAL Segment Switches

WAL segment switches occur when one of the following events happens:

1. WAL segment is filled up.
2. The function `pg_switch_xlog` is called.
3. The `archive_mode` parameter is enabled and the `archive_timeout` parameter has expired.

When a WAL segment file is switched, it is usually recycled (renamed and reused) for future use. However, it may be removed later if it is not needed.

### 9.9.2. WAL Segment Management

Whenever a checkpoint starts, PostgreSQL estimates and prepares the number of WAL segment files that will be needed for the next checkpoint cycle. This estimate is based on the number of WAL segment files that were consumed in previous checkpoint cycles.

The number of WAL segment files is counted from the segment that contains the prior REDO point, and the value must be between the `min_wal_size` parameter (which default to 80 MB, or 5 files) and the `max_wal_size` parameter (which default to 1 GB, or 64 files).

If a checkpoint starts, PostgreSQL will keep or recycle the necessary WAL segment files, and remove any unnecessary files.

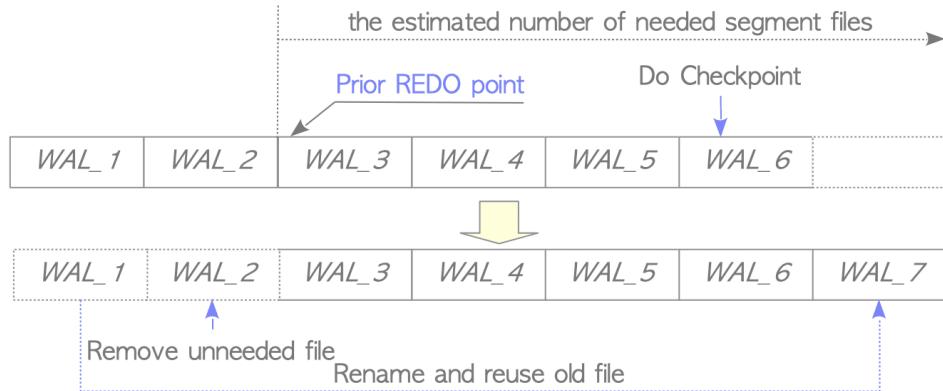
A specific example is shown in Fig. 9.17. Assuming that there are six WAL segment files before a checkpoint starts, *WAL\_3* contains the prior REDO point (in versions 10 or earlier; in versions 11 or later, the REDO point), and PostgreSQL estimates that five files will be needed. In this case, *WAL\_1* will be renamed as *WAL\_7* for recycling and *WAL\_2* will be removed.



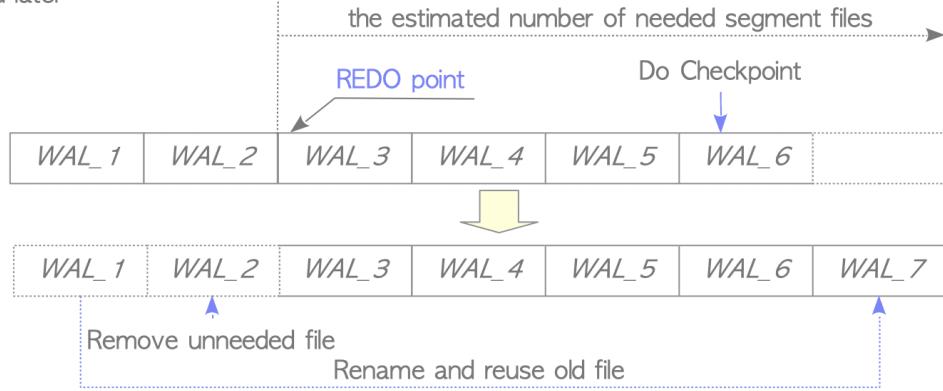
The files older than the one that contains the prior REDO point can be removed, because, as is clear from the recovery mechanism described in Section 9.8, they would never be used.

**Fig. 9.17. Recycling and removing WAL segment files at a checkpoint.**

Version 9.5, 9.6, and 10

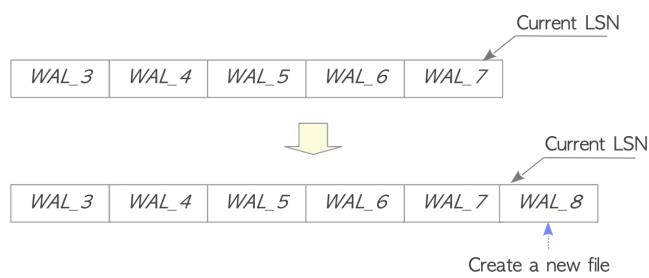


Version 11 and later



If more WAL segment files are required due to a spike in WAL activity, new WAL segment files will be created while the total size of the WAL segment files is less than the *max\_wal\_size* parameter. For example, in Fig. 9.18, if *WAL\_7* has been filled up, *WAL\_8* will be newly created.

**Fig. 9.18. Creating WAL segment file.**

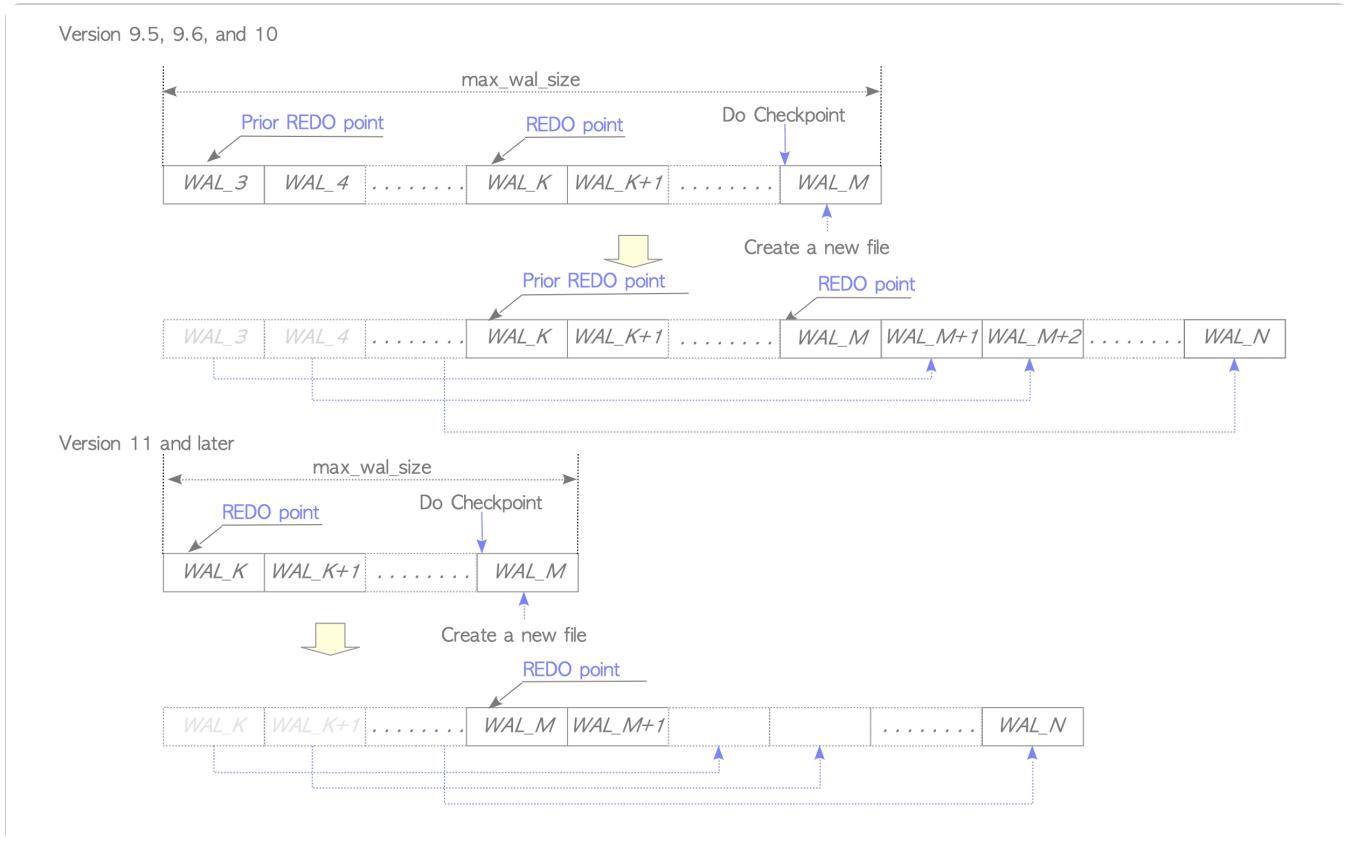


The number of WAL segment files adapts to the server activity. If the amount of WAL data writing has been constantly increasing, the estimated number of WAL segment files, as well as the total size of the WAL segment files, will gradually increase. In the opposite case (i.e., the amount of WAL data writing has decreased), these will decrease.

If the total size of the WAL segment files exceeds the *max\_wal\_size* parameter, a checkpoint will be started. Figure 9.19 illustrates this situation. By checkpointing, a new REDO point will be created and the prior REDO point will be discarded; then, unnecessary old WAL segment files will be

recycled. In this way, PostgreSQL will always keep just the WAL segment files that are needed for database recovery.

**Fig. 9.19. Checkpointing and recycling WAL segment files.**



The `wal_keep_size` (or `wal_keep_segments` in versions 12 or earlier) and the replication slot feature also affect the number of WAL segment files.

## 9.10. Continuous Archiving and Archive Logs

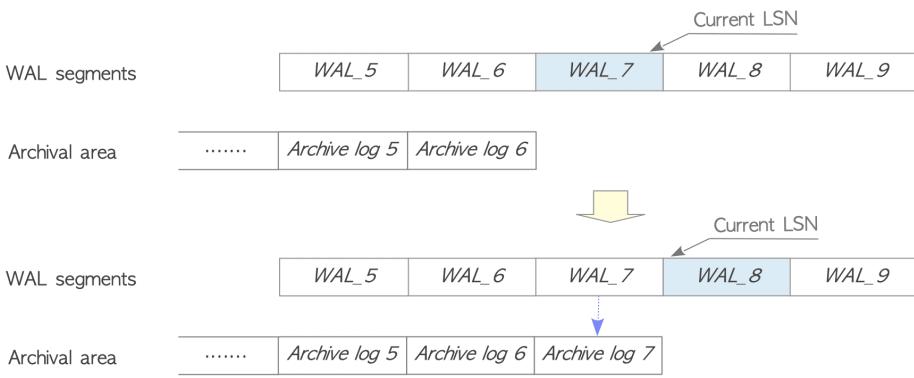
**Continuous archiving** is a feature that copies WAL segment files to an archival area at the time when a WAL segment switch occurs. It is performed by the *archiver (background)* process. The copied file is called an **archive log**. This feature is typically used for hot physical backup and PITR (Point-in-Time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

The path to the archival area is set by the `archive_command` configuration parameter. For example, the following parameter would copy WAL segment files to the directory `'/home/postgres/archives/'` every time a segment switch occurs:

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

where, the `%p` placeholder represents the copied WAL segment, and the `%f` placeholder represents the archive log.

**Fig. 9.20. Continuous archiving.**



When the WAL segment file *WAL\_7* is switched, the file is copied to the archival area as *Archive log 7*.

The *archive\_command* parameter can be set to any Unix command or tool. This means that you can use the `scp` command or any other file backup tool to transfer the archive logs to another host, instead of using a simple copy command.

## archive\_library

In versions 14 or earlier, continuous archiving could only use shell commands. In version 15, PostgreSQL introduced a loadable library feature that allows you to achieve continuous archiving using a library. For more information, see the `archive_library` and `basic_archive` documentation.



PostgreSQL does not automatically clean up created archive logs. Therefore, you must properly manage the logs when using this feature. If you do nothing, the number of archive logs will continue to grow.

The `pg_archivecleanup` utility is one of the useful tools for managing archive log files.

You can also use the **find** command to delete archive logs. For example, the following command would delete all archive logs that were created more than three days ago:

```
$ find /home/postgres/archives -mtime +3d -exec rm -f {} \;
```

# Chapter 10

## Base Backup & Point-in-Time Recovery

Online database backup can be roughly classified into two categories: logical and physical backups. While both have advantages and disadvantages, one disadvantage of logical backups is that they can be very time-consuming. In particular, it can take a long time to make a backup of a large database, and even longer to restore the database from the backup data. On the other hand, physical backups can be made and restored much more quickly, making them a very important and useful feature in practical systems.

In PostgreSQL, online physical full backups have been available since version 8.0. A snapshot of a running whole database cluster (i.e., physical backup data) is known as a **base backup**.

**Point-in-Time Recovery (PITR)**, which has also been available since version 8.0, is the feature to restore a database cluster to any point in time using a *base backup* and *archive logs* created by the continuous archiving feature. For example, if you make a critical mistake (such as truncating all tables), this feature can be used to restore the database to the point just before the mistake was made.

In this chapter, following topics are described:

- What is a base backup?
- How does PITR work?
- What is a timelineId?
- What is a timeline history file?



In versions 7.4 or earlier, PostgreSQL had supported only logical backups (logical full and partial backups, and data exports).

### 10.1. Base Backup

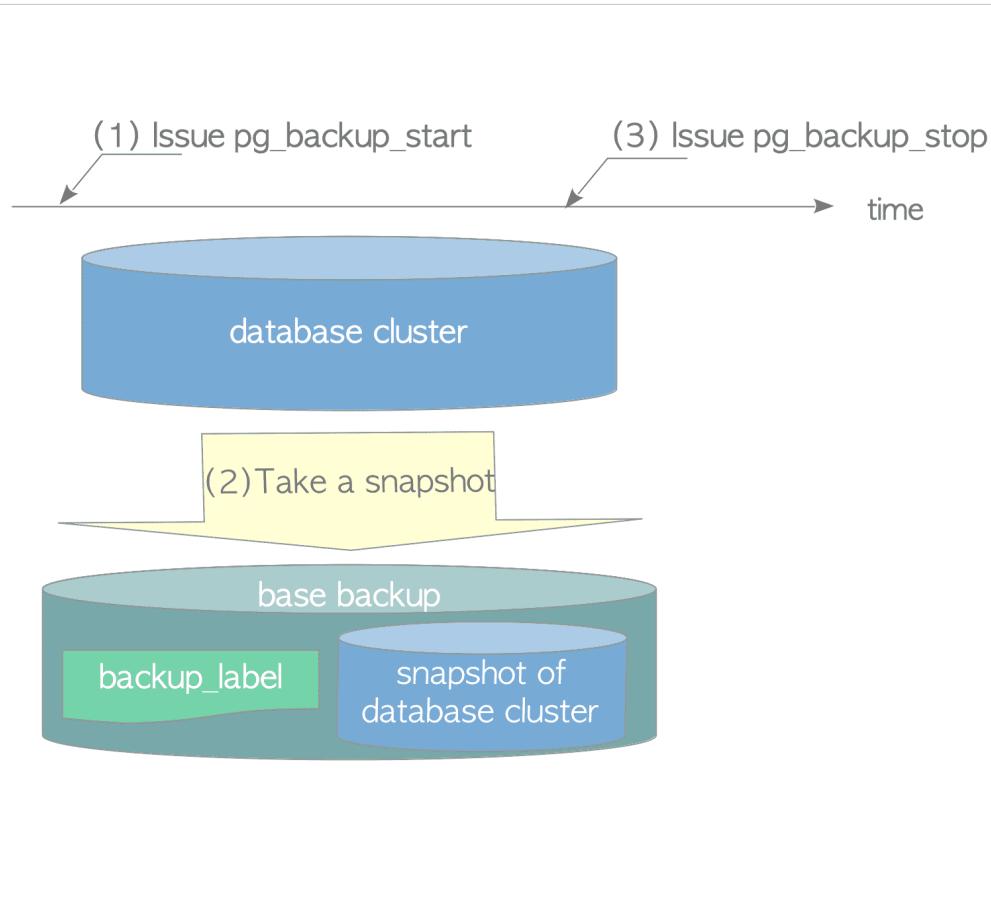
First of all, the standard procedure to make a base backup using the low-level commands is as follows:

- (1) Issue the `pg_backup_start` command (versions 14 or earlier, `pg_start_backup`).
- (2) Take a snapshot of the database cluster using the archiving command of your choice.
- (3) Issue the `pg_backup_stop` command (versions 14 or earlier, `pg_stop_backup`).

This simple procedure is easy for database administrators to use because it requires no special tools other than common tools such as the `cp` command or a similar archiving tool. In addition, this procedure does not require table locks, so all users can continue to issue queries without being affected by the backup operation. This is a significant advantage over other major open source RDBMSs.

A simpler way to make a base backup is to use the `pg_basebackup` utility, which internally issues the low-level commands described above.

**Fig. 10.1. Making a base backup.**



Because the `pg_backup_start` and `pg_backup_stop` commands are so important to understanding PITR, we will explore them in more detail in the following subsections.



The `pg_backup_start` and `pg_backup_stop` commands are defined here: <src/backend/access/transam/xlogfuncs.c>.

### 10.1.1. `pg_backup_start` (Ver.14 or earlier, `pg_start_backup`)

The `pg_backup_start` command prepares for making a base backup. As discussed in Section 9.8, the recovery process starts from a REDO point, so the `pg_backup_start` command must do a checkpoint to explicitly create a REDO point at the start of making a base backup. Moreover, the checkpoint location of its checkpoint must be saved in a file other than `pg_control` because regular

checkpoints might be done a number of times during the backup. Therefore, the `pg_backup_start` performs the following four operations:

1. Force the database into full-page write mode.
2. Switch to the current WAL segment file (versions 8.4 or later).
3. Do a checkpoint.
4. Create a `backup_label` file – This file, created in the top level of the base directory, contains essential information about base backup itself, such as the checkpoint location of this checkpoint.

The third and fourth operations are the heart of this command. The first and second operations are performed to recover a database cluster more reliably.

A `backup_label` file contains the following six items (versions 11 or later, seven items):

- CHECKPOINT LOCATION – This is the LSN location where the checkpoint created by this command has been recorded.
- START WAL LOCATION – This is **not** used with PITR, but used with the streaming replication, which is described in Chapter 11. It is named 'START WAL LOCATION' because the standby server in replication-mode reads this value only once at initial startup.
- BACKUP METHOD – This is the method used to make this base backup.
- BACKUP FROM – This shows whether this backup is taken from the primary or standby server.
- START TIME – This is the timestamp when the `pg_backup_start` command was executed.
- LABEL – This is the label specified at the `pg_backup_start` command.
- START TIMELINE – This is the timeline that the backup started. This is for a sanity check and has been introduced in version 11.

### `backup_label`

An actual example of a `backup_label` file in version 16, which is taken by using `pg_basebackup`, is shown below:

```
postgres> cat /usr/localpgsql/data/backup_label
START WAL LOCATION: 0/1B000028 (file 000000010000000000000001B)
CHECKPOINT LOCATION: 0/1B000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-10-26 11:45:19 GMT
LABEL: pg_basebackup base backup
START TIMELINE: 1
```

As you may imagine, when you recover a database using this base backup, PostgreSQL takes the 'CHECKPOINT LOCATION' from the `backup_label` file to read the checkpoint record from the appropriate archive log. It then gets the REDO point from the record and starts the recovery process. (The details will be described in the next section.)

## **10.1.2. `pg_backup_stop` (Ver.14 or earlier, `pg_stop_backup`)**

The `pg_backup_stop` command performs the following five operations to complete the backup:

1. Reset to *non-full-page writes* mode if it has been forcibly changed by the `pg_backup_start` command.
2. Write a XLOG record of backup end.

3. Switch the WAL segment file.
4. Create a *backup history file*. This file contains the contents of the *backup\_label* file and the timestamp that the *pg\_backup\_stop* command was executed.
5. Delete the *backup\_label* file. The *backup\_label* file is required for recovery from the base backup, but once copied, it is not necessary in the original database cluster.



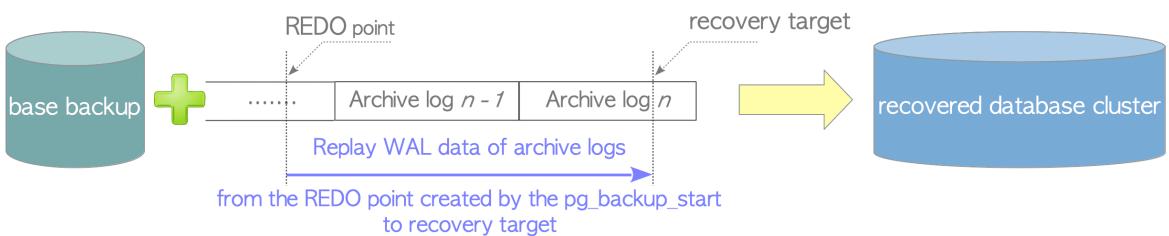
The *naming method for backup history file* is shown below.

```
{WAL segment}.{offset value at the time the base backup was started}.backup
```

## 10.2. How Point-in-Time Recovery Works

Figure 10.2 shows the basic concept of PITR. In PITR mode, PostgreSQL replays the WAL data of the archive logs on the base backup, from the REDO point created by the *pg\_backup\_start* up to the point you want to recover. In PostgreSQL, the point to be recovered is referred to as a **recovery target**.

**Fig. 10.2. Basic concept of PITR.**



Here is the description of how PITR works.

Suppose that you made a mistake at 12:05 GMT of 26 October, 2023. You should remove the database cluster and restore the new one using the base backup you made before that.

At first, you need to set the command of the *restore\_command* parameter, and also set the time of the *recovery\_target\_time* parameter to the point you made the mistake (in this case, 12:05 GMT) in a *postgresql.conf* (versions 12 or later) or *recovery.conf* (versions 11 or earlier).

```
# Place archive logs under /mnt/server/archivedir directory.
restore_command = 'cp /mnt/server/archivedir/%f %p'
recovery_target_time = "2023-10-26 12:05 GMT"
```

When PostgreSQL starts up, it enters into PITR mode if there is a *recovery.signal* (versions 12 or later) or a *recovery.conf* (versions 11 or earlier), and a *backup\_label* in the database cluster.

### i recovery.conf

The *recovery.conf* file has been abolished in version 12, and all recovery-related parameters should be written in *postgresql.conf*. See the official document in detail.

In versions 12 or later, when you restore your server from a basebackup, you need to create an empty file called **recovery.signal** in the database cluster directory.

```
$ touch /usr/local/pgsql/data/recovery.signal
```

The PITR (Point-in-Time Recovery) process is almost the same as the normal recovery process described in Chapter 9. The only differences are:

1. Where are WAL segments/Archive logs read from?
  - Normal recovery mode – from the pg\_wal subdirectory (in versions 9.6 or earlier, pg\_xlog subdirectory) under the base directory.
  - PITR mode – from an archival directory set in the configuration parameter archive\_command.
2. Where is the checkpoint location read from?
  - Normal recovery mode – from the *pg\_control file*.
  - PITR mode – from a *backup\_label file*.

The outline of PITR process is as follows:

- (1) PostgreSQL reads the value of '*CHECKPOINT LOCATION*' from the backup\_label file using the internal function *read\_backup\_label* to find the REDO point.
- (2) PostgreSQL reads some values of parameters from the postgresql.conf (versions 12 or later) or recovery.conf (versions 11 or earlier), such as *restore\_command* and *recovery\_target\_time*.
- (3) PostgreSQL starts replaying WAL data from the REDO point, which can be easily obtained from the value of '*CHECKPOINT LOCATION*'. The WAL data are read from archive logs that are copied from the archival area to a temporary area by executing the command written in the *restore\_command* parameter. (The copied log files in the temporary area are removed after use.) In this example, PostgreSQL reads and replays WAL data from the REDO point to the one before the timestamp '2023-10-26 12:05:00' because the *recovery\_target\_time* parameter is set to this timestamp. If a recovery target is not set to the postgresql.conf (versions 12 or later) or recovery.conf (versions 11 or earlier), PostgreSQL will replay until the end of the archiving logs.
- (4) When the recovery process completes, a **timeline history file**, such as '00000002.history', is created in the pg\_wal subdirectory (in versions 9.6 or earlier, pg\_xlog subdirectory). If archiving log feature is enabled, the same named file is also created in the archival directory. The contents and role of this file are described in the following sections.

The records of commit and abort actions contain the timestamp at which each action has done (XLOG data portion of both actions are defined in *xl\_xact\_commit* and *xl\_xact\_abort* respectively). Therefore, if a target time is set to the parameter *recovery\_target\_time*, PostgreSQL may select whether to continue recovery or not, whenever it replays XLOG record of either commit or abort action. When XLOG record of each action is replayed, PostgreSQL compares the target time and each timestamp written in the record; and if the timestamp exceed the target time, PITR process will be finished.



The function *read\_backup\_label* is defined in src/backend/access/transam/xlog.c.  
The structure *xl\_xact\_commit* and *xl\_xact\_abort* are defined in src/include/access/xact.h.

## ❸ Why can we use common archiving tools to make a base backup?

The recovery process is a process of restoring a database cluster to a consistent state, even if the cluster is inconsistent. PITR is based on the recovery process, so it can recover a database cluster even if the base backup is a bunch of inconsistent files. This is why we can use common archiving tools without the need for a file system snapshot capability or a special tool.

## 10.3. timelineld and Timeline History File

A **timeline** in PostgreSQL is used to distinguish between the original database cluster and the recovered ones. It is a central concept of PITR. In this section, two things associated with the timeline are described: *timelineld* and *timeline history files*.

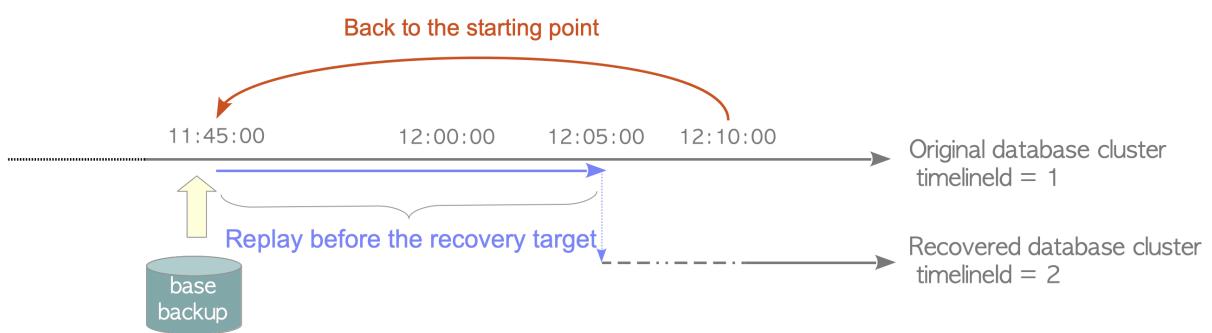
### 10.3.1. timelineld

Each timeline is given a corresponding **timelineld**, a 4-byte unsigned integer starting at 1.

An individual timelineld is assigned to each database cluster. The timelineld of the original database cluster created by the initdb utility is 1. Whenever a database cluster recovers, the timelineld is increased by 1. For example, in the example of the previous section, the timelineld of the cluster recovered from the original one is 2.

Figure 10.3 illustrates the PITR process from the viewpoint of the timelineld. First, we remove our current database cluster and restore the base backup made in the past, in order to go back to the starting point of recovery. This situation is represented by the red arrow curve in the figure. Next, we start the PostgreSQL server, which replays WAL data in the archive logs from the REDO point created by the `pg_backup_start` until the recovery target by tracing along the initial timeline (timelineld 1). This situation is represented by the blue arrow line in the figure. Then, a new timelineld 2 is assigned to the recovered database cluster and PostgreSQL runs on the new timeline.

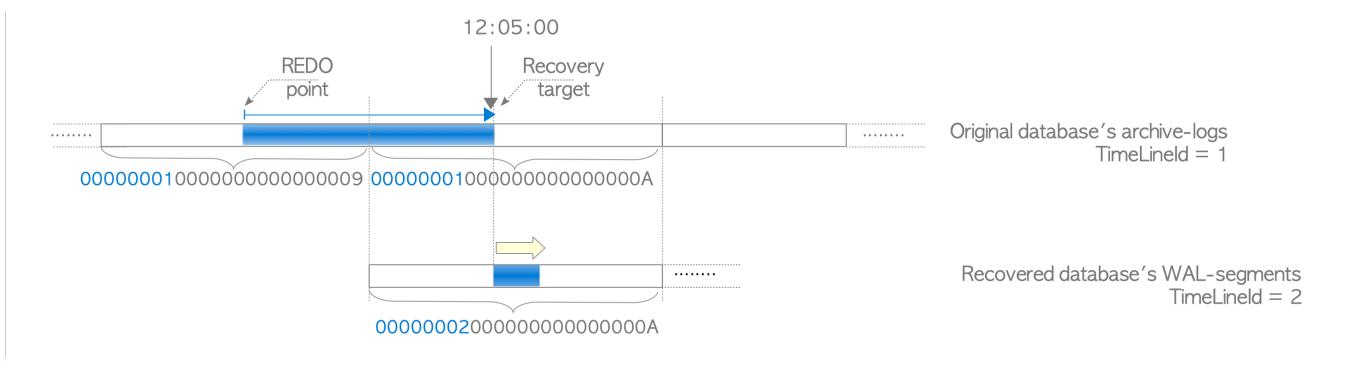
**Fig. 10.3. Relation of timelineld between an original and a recovered database clusters.**



As briefly mentioned in Chapter 9, the first 8 digits of the WAL segment filename are equal to the timelineld of the database cluster that created the segment. When the timelineld is changed, the WAL segment filename will also be changed.

Focusing on WAL segment files, the recovery process can be described again. Suppose that we recover the database cluster using two archive logs '`000000010000000000000009`' and '`00000001000000000000000A`'. The newly recovered database cluster is assigned the timelineId 2, and PostgreSQL creates the WAL segment from '`00000002000000000000000A`'. Figure 10.4 shows this situation.

**Fig. 10.4. Relation of WAL segment files between an original and a recovered database clusters.**



### 10.3.2. Timeline History File

When a PITR process completes, a timeline history file with names like '`00000002.history`' is created under the archival directory and the `pg_xlog` subdirectory (in versions 10 or later, `pg_wal` subdirectory). This file records which timeline it branched off from and when.

The naming rule of this file is shown below:

"8-digit new timelineId".history

The timeline history file contains at least one line, and each line is composed of the following three items:

- timelineId – The timelineId of the archive logs used to recover.
- LSN – The LSN location where the WAL segment switches happened.
- reason – A human-readable explanation of why the timeline was changed.

A specific example is shown below:

```
postgres> cat /home/postgres/archivelogs/00000002.history
1          0/A000198      before 2023-10-26 12:05:00.861324+00
```

Meaning as follows:

The database cluster (timelineId=2) is based on the base backup whose timelineId is 1, and is recovered in the time just *before* '`2023-10-26 12:05:00.861324+00`' by replaying the archive logs until the `0/A000198`.

In this way, each timeline history file tells us a complete history of the individual recovered database cluster. Moreover, it is also used in the PITR process itself. The details are explained in the next section.



The timeline history file format is changed in version 9.3. The formats of versions 9.3 or later and earlier are shown below, but not in detail.

Later version 9.3:

```
timelineId LSN "reason"
```

Until version 9.2:

```
timelineId WAL_segment "reason"
```

## 10.4. Point-in-Time Recovery with Timeline History File

The timeline history file plays an important role in the second and subsequent PITR processes. By trying a second time recovery, we will explore how it is used.

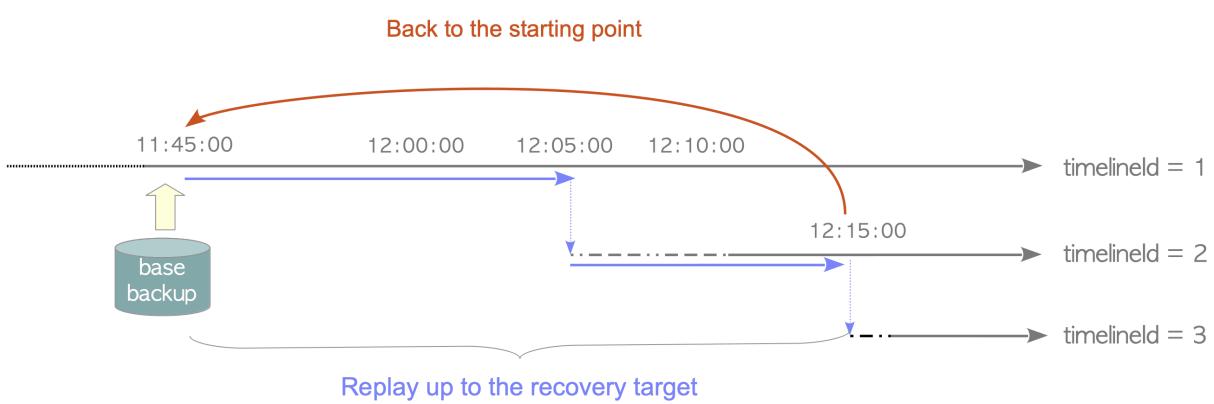
Again, suppose that you made a mistake at 12:15:00 in the recovered database cluster whose timelineId is 2. In this case, to recover the database cluster, you should create a new recovery.conf file as shown below:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
recovery_target_time = "2023-10-26 12:15:00 GMT"
recovery_target_timeline = 2
```

The parameter *recovery\_target\_time* sets the time you made the new mistake, and the parameter *recovery\_target\_timeline* is set at '2' in order to recover along its timeline.

Restart the PostgreSQL server and enter PITR mode to recover the database at the target time along the timelineId 2. See Fig. 10.5.

**Fig. 10.5. Recover the database at 12:15:00 along the timelineId 2.**



- (1) PostgreSQL reads the value of 'CHECKPOINT LOCATION' from the backup\_label file.
- (2) Some values of parameters are read from the recovery.conf; in this example, *restore\_command*, *recovery\_target\_time*, and *recovery\_target\_timeline*.
- (3) PostgreSQL reads the timeline history file '00000002.history' which is corresponding to the value of the parameter *recovery\_target\_timeline*.
- (4) PostgreSQL does replaying WAL data by the following steps:
  1. From the REDO point to the LSN '0/A000198', which is written in the 00000002.history file, PostgreSQL reads and replays WAL data of appropriate archive logs whose timelineId is

- 1.
  2. From the one after LSN '0/A000198' to the one before the timestamp '2023-10-26 12:15:00', PostgreSQL reads and replays WAL data (of appropriate archive logs) whose timelineId is 2.
- (5) When the recovery process completes, the current timelineId will advance to 3, and a new timeline history file named *00000003.history* is created in the pg\_wal subdirectory (pg\_xlog if versions 9.6 or earlier) and the archival directory.

```
postgres> cat /home/postgres/archivelogs/00000003.history
1      0/A000198  before 2023-10-26 12:05:00.861324+00
2      0/B000078  before 2023-10-26 12:15:00.927133+00
```

When you do PITR more than once, you should explicitly set a timelineId for using the appropriate timeline history file.

In this way, timeline history files are not only history logs of database cluster, but also the recovery instruction documents for PITR process.

# Chapter 11

## Streaming Replication

---

**S**ynchronous streaming replication was implemented in version 9.1. It is a single-master-multi-slaves type of replication, where the terms "master" and "slaves" are usually referred to as **primary** and **standbys** respectively.

This native replication feature is based on log shipping, a general replication technique in which the primary server continuously sends **WAL (Write-Ahead Log) data** to the standby servers, which then replay the received data immediately.

This chapter focuses on how streaming replication works and covers the following topics:

- How streaming replication starts up
- How data is transferred between the primary and standby servers
- How the primary server manages multiple standby servers
- How the primary server detects failures of standby servers



Although the first replication feature, which was only for asynchronous replication, was implemented in version 9.0, it was replaced with a new implementation (currently in use) for synchronous replication in version 9.1.

### 11.1. Starting the Streaming Replication

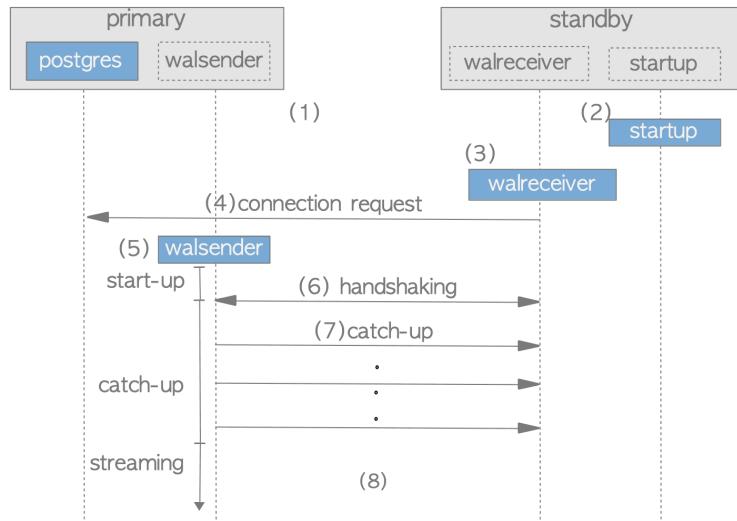
In streaming replication, three types of processes work cooperatively:

- A **walsender** process on the primary server sends WAL (Write-Ahead Log) data to the standby server.
- A **walreceiver** process on the standby server receives and replays the WAL data.
- A **startup** process on the standby server starts the walreceiver process.

The walsender and walreceiver communicate using a single TCP connection.

The startup sequence of streaming replication is shown in Figure 11.1:

**Fig. 11.1. SR startup sequence.**



- (1) Start the primary and standby servers.
- (2) The standby server starts the startup process.
- (3) The standby server starts a walreceiver process.
- (4) The walreceiver sends a connection request to the primary server. If the primary server is not running, the walreceiver sends these requests periodically.
- (5) When the primary server receives a connection request, it starts a walsender process and a TCP connection is established between the walsender and walreceiver.
- (6) The walreceiver sends the latest LSN (Log Sequence Number) of standby's database cluster. This is known as **handshaking** in the field of information technology.
- (7) If the standby's latest LSN is less than the primary's latest LSN (Standby's LSN < Primary's LSN), the walsender sends WAL data from the former LSN to the latter LSN. These WAL data are provided by WAL segments stored in the primary's pg\_wal subdirectory (in versions 9.6 or earlier, pg\_xlog). The standby server then replays the received WAL data. In this phase, the standby catches up with the primary, so it is called **catch-up**.
- (8) Streaming Replication begins to work.

Each walsender process keeps a state that is appropriate for the working phase of the connected walreceiver or application. The following are the possible states of a walsender process:

- start-up – From starting the walsender to the end of handshaking. See Figs. 11.1(5)–(6).
- catch-up – During the catch-up phase. See Fig. 11.1(7).
- streaming – While Streaming Replication is working. See Fig. 11.1(8).
- backup – During sending the files of the whole database cluster for backup tools such as `pg_basebackup` utility.

The `pg_stat_replication` view shows the state of all running walsenders. An example is shown below:

```
testdb=# SELECT application_name, state FROM pg_stat_replication;
application_name | state
-----+-----
standby1      | streaming
standby2      | streaming
pg_basebackup | backup
(3 rows)
```

As shown in the above result, two walsenders are running to send WAL data for the connected standby servers, and another one is running to send all files of the database cluster for `pg_basebackup` utility.

## ⌚ What will happen if a standby server restarts after a long time in the stopped condition?

In versions 9.3 or earlier, if the primary's WAL segments required by the standby server have already been recycled, the standby cannot catch up with the primary server. There is no reliable solution for this problem, but only to set a large value to the configuration parameter *wal\_keep\_segments* to reduce the possibility of the occurrence. This is a stopgap solution.

In versions 9.4 or later, this problem can be prevented by using *replication slot*. A replication slot is a feature that expands the flexibility of the WAL data sending, mainly for the *logical replication*, which also provides the solution to this problem – the WAL segment files that contain unsent data under the *pg\_wal* (or *pg\_xlog* if versions 9.6 or earlier) can be kept in the replication slot by pausing recycling process. Refer the official document for detail.

## 11.2. How to Conduct Streaming Replication

Streaming replication has two aspects: log shipping and database synchronization. Log shipping is the main aspect of streaming replication, as the primary server sends WAL (Write-Ahead Log) data to the connected standby servers whenever they are written. Database synchronization is required for synchronous replication, where the primary server communicates with each standby server to synchronize their database clusters.

To accurately understand how streaming replication works, we need to understand how one primary server manages multiple standby servers. We will start with the simple case (i.e., single-primary single-standby system) in this section, and then discuss the general case (single-primary multi-standby system) in the next section.

### 11.2.1. Communication Between a Primary and a Synchronous Standby

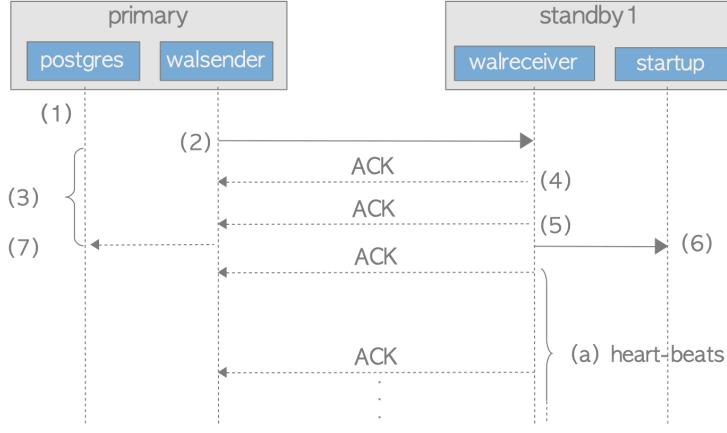
Assume that the standby server is in the synchronous replication mode, but the configuration parameter *hot\_standby* is disabled and *wal\_level* is '*replica*'. The main parameter of the primary server is shown below:

```
synchronous_standby_names = 'standby1'  
hot_standby = off  
wal_level = replica
```

Additionally, among the three triggers to write the WAL data mentioned in Section 9.5, we focus on the transaction commits here.

Suppose that one backend process on the primary server issues a simple INSERT statement in autocommit mode. The backend starts a transaction, issues an INSERT statement, and then commits the transaction immediately. Let's explore further how this commit action will be completed. See the following sequence diagram in Fig. 11.2:

**Fig. 11.2. Streaming Replication's communication sequence diagram.**



- (1) The backend process writes and flushes WAL data to a WAL segment file by executing the functions *XLogInsert()* and *XLogFlush()*.
- (2) The walsender process sends the WAL data written into the WAL segment to the walreceiver process.
- (3) After sending the WAL data, the backend process continues to wait for an ACK response from the standby server. More precisely, the backend process gets a latch by executing the internal function *SyncRepWaitForLSN()*, and waits for it to be released.
- (4) The walreceiver on the standby server writes the received WAL data into the standby's WAL segment using the *write()* system call, and returns an ACK response to the walsender.
- (5) The walreceiver flushes the WAL data to the WAL segment using the system call such as *fsync()*, returns another ACK response to the walsender, and informs the startup process about WAL data updated.
- (6) The startup process replays the WAL data, which has been written to the WAL segment.
- (7) The walsender releases the latch of the backend process on receiving the ACK response from the walreceiver, and then, the backend process's commit or abort action will be completed. The timing for latch-release depends on the parameter *synchronous\_commit*. It is '*on*' (default), the latch is released when the ACK of step (5) received, whereas it is '*remote\_write*', the latch is released when the ACK of step (4) is received.

Each ACK response informs the primary server of the internal information of standby server. It contains four items below:

```

XLogWalRcvSendReply(void)@src/backend/replication/walreceiver.c
/*
 * Construct a new message */
writePtr = LogstreamResult.Write;
flushPtr = LogstreamResult.Flush;
applyPtr = GetXLogReplayRecPtr(NULL);

resetStringInfo(&reply_message);
pq_sendbyte(&reply_message, 'r');
pq_sendint64(&reply_message, writePtr);
pq_sendint64(&reply_message, flushPtr);
pq_sendint64(&reply_message, applyPtr);
pq_sendint64(&reply_message, GetCurrentTimestamp());
pq_sendbyte(&reply_message, requestReply ? 1 : 0);

```

- The LSN location where the latest WAL data has been written.
- The LSN location where the latest WAL data has been flushed.
- The LSN location where the latest WAL data has been replayed in the startup process.
- The timestamp when this response has been sent.

The walreceiver returns ACK responses not only when WAL data have been written and flushed, but also periodically as a heartbeat from the standby server. The primary server therefore always has an accurate understanding of the status of all connected standby servers.

The LSN-related information of the connected standby servers can be displayed by issuing the queries shown below:

```
testdb=# SELECT application_name AS host,
    write_location AS write_LSN, flush_location AS flush_LSN,
    replay_location AS replay_LSN FROM pg_stat_replication;

 host | write_lsn | flush_lsn | replay_lsn
-----+-----+-----+-----+
 standby1 | 0/5000280 | 0/5000280 | 0/5000280
 standby2 | 0/5000280 | 0/5000280 | 0/5000280
(2 rows)
```



The heartbeat interval is set to the parameter *wal\_receiver\_status\_interval*, which is 10 seconds by default.

## 11.2.2. Behavior When a Failure Occurs

In this subsection, I describe how the primary server behaves when a synchronous standby server fails, and how to deal with the situation.

Even if a synchronous standby server fails and is no longer able to return an ACK response, the primary server will continue to wait for responses forever. This means that running transactions cannot commit and subsequent query processing cannot be started. In other words, all primary server operations are effectively stopped. (Streaming replication does not support a function to automatically revert to asynchronous mode after a timeout.)

There are two ways to avoid such situation. One is to use multiple standby servers to increase system availability. The other is to manually switch from synchronous to *asynchronous* mode by performing the following steps:

- (1) Set the parameter *synchronous\_standby\_names* to an empty string.

```
synchronous_standby_names = ""
```

- (2) Execute the pg\_ctl command with *reload* option.

```
postgres> pg_ctl -D $PGDATA reload
```

This procedure does not affect connected clients. The primary server will continue to process transactions and all sessions between clients and their respective backend processes will be maintained.

## 11.3. Managing Multiple-Standby Servers

In this section, I describe the way streaming replication works with multiple standby servers.

### 11.3.1. sync\_priority and sync\_state

The primary server assigns the *sync\_priority* and *sync\_state* attributes to all managed standby servers, and treats each standby server according to its respective values. (The primary server assigns these values even if it manages just one standby server; this was not mentioned in the previous section.)

The *sync\_priority* attribute indicates the priority of the standby server in synchronous mode. The lower the value, the higher the priority. The special value 0 means that the standby server is '*in asynchronous mode*'. The priorities of the standby servers are assigned in the order listed in the primary server's configuration parameter *synchronous\_standby\_names*. For example, in the following configuration, the priorities of standby1 and standby2 are 1 and 2, respectively.

```
synchronous_standby_names = 'standby1,standby2'
```

(Standby servers that are not listed in this parameter are in asynchronous mode and have a priority of 0.)

*sync\_state* is the state of the standby server. The *sync\_state* attribute indicates the state of the standby server. It can be one of the following values:

- **sync:** The standby server is in synchronous mode and is the highest priority standby server that is currently working.
- **potential:** The standby server is in synchronous mode and is a lower priority standby server that is currently working. If the current sync standby server fails, this standby server will be promoted to sync state.
- **async:** The standby server is in asynchronous mode. (It will never be in 'sync' or 'potential' mode.)

The priority and state of the standby servers can be shown by issuing the following query:

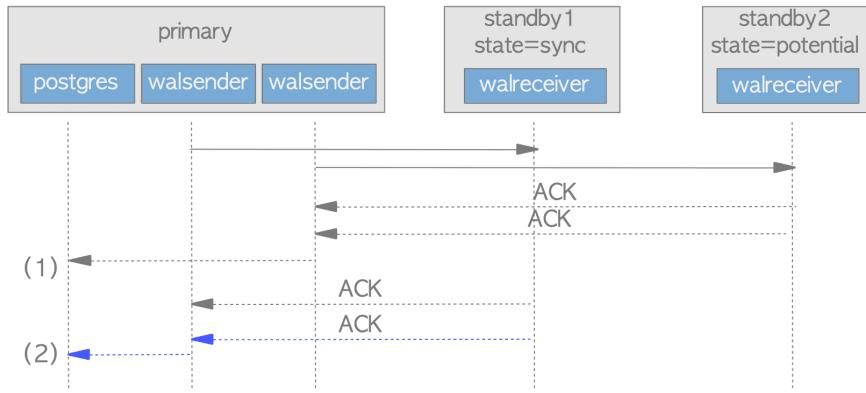
```
testdb=# SELECT application_name AS host,
           sync_priority, sync_state FROM pg_stat_replication;
   host   | sync_priority | sync_state
-----+-----+-----+
standby1 |         1 | sync
standby2 |         2 | potential
(2 rows)
```

### 11.3.2. How the Primary Manages Multiple-standbys

The primary server waits for ACK responses from the synchronous standby server alone. In other words, the primary server confirms only the synchronous standby's writing and flushing of WAL data. Streaming replication, therefore, ensures that only the synchronous standby is in a consistent and synchronous state with the primary.

Figure 11.3 shows the case in which the ACK response of the potential standby has been returned earlier than that of the primary standby. In this case, the primary server does not complete the commit action of the current transaction and continues to wait for the primary's ACK response. When the primary's response is received, the backend process releases the latch and completes the current transaction processing.

**Fig. 11.3. Managing multiple standby servers.**



The sync\_state of standby1 and standby2 are 'sync' and 'potential' respectively. (1) The primary's backend process continues to wait for an ACK response from the synchronous standby server, even though it has received an ACK response from the potential standby server. (2) After receiving the ACK response from the synchronous standby server, the primary's backend process releases the latch and completes the current transaction processing.

In the opposite case (i.e., the primary's ACK response has been returned earlier than the potential's), the primary server immediately completes the commit action of the current transaction without ensuring if the potential standby writes and flushes WAL data or not.

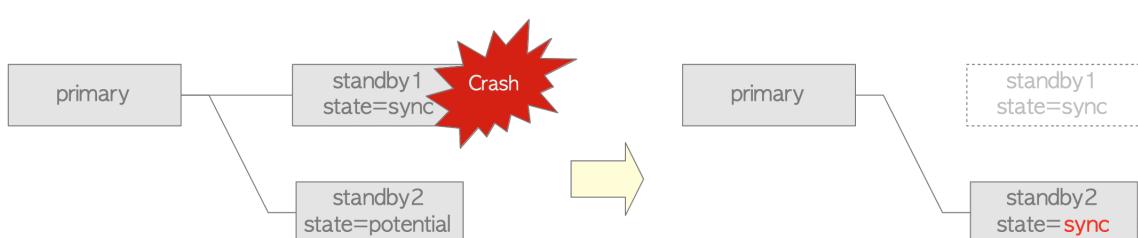
### 11.3.3. Behavior When a Failure Occurs

Once again, let's see how the primary server behaves when a standby server has failed.

When either a potential or an asynchronous standby server has failed, the primary server terminates the walsender process connected to the failed standby and continues all processing. In other words, transaction processing on the primary server would not be affected by the failure of either type of standby server.

When a synchronous standby server has failed, the primary server terminates the walsender process connected to the failed standby, and replaces the synchronous standby with the highest priority potential standby. See Fig. 11.4. In contrast to the failure described above, query processing on the primary server will be paused from the point of failure to the replacement of the synchronous standby. (Therefore, failure detection of the standby server is a very important function to increase the availability of the replication system. Failure detection will be described in the next section.)

**Fig. 11.4. Replacing of synchronous standby server.**



In any case, if one or more standby servers are running in synchronous mode, the primary server keeps only one synchronous standby server at all times, and the synchronous standby server is always in a consistent and synchronous state with the primary.

## 11.4. Detecting Failures of Standby Servers

Streaming replication uses two common failure detection procedures that do not require any special hardware.

### 1. Failure detection of standby server process:

- When a connection drop between the walsender and walreceiver is detected, the primary server *immediately* determines that the standby server or walreceiver process is faulty.
- When a low-level network function returns an error by failing to write or read the socket interface of the walreceiver, the primary server also *immediately* determines its failure.

### 2. Failure detection of hardware and networks:

- If a walreceiver does not return anything within the time set for the parameter `wal_sender_timeout` (default 60 seconds), the primary server determines that the standby server is faulty.
- In contrast to the failure described above, it takes a certain amount of time, up to `wal_sender_timeout` seconds, to confirm the standby's death on the primary server even if a standby server is no longer able to send any response due to some failures (e.g., standby server's hardware failure, network failure, etc.).

Depending on the type of failure, it can usually be detected immediately after the failure occurs. However, there may be a time lag between the occurrence of the failure and its detection. In particular, if the latter type of failure occurs in a synchronous standby server, all transaction processing on the primary server will be stopped until the failure of the standby is detected, even if multiple potential standby servers may have been working.