What's New

- 14.Dec.2024: Added Conflicts in Section 11.2.4.
- 13.Dec.2024: Added Replication Slots in Section 11.4.
- 30.Nov.2024: Added Parallel Query in Section 3.7.
- 28.Oct.2024: Added Incremental Backup in Section 10.5.

› Change History (since 3rd June, 2018)

The Internals of PostgreSQL

PostgreSQL is a well-designed, open-source multi-purpose relational database system which is widely used throughout the world.

It is one huge system with the integrated subsystems, each of which has a particular complex feature and works cooperatively with each other. Although understanding of the internal mechanism is crucial for both administration and integration using PostgreSQL, its hugeness and complexity make it difficult.

The main purposes of this document are to explain how each subsystem works, and to provide the whole picture of PostgreSQL.

This document covers versions 17 and earlier.

Some academic papers have referred to this document. The Chinese version of this document was published in June 2019.

Contents

Author

Hironobu SUZUKI

I am a software programmer/engineer, the author of:

- The Internals of PostgreSQL
- The Engineer's Guide To Deep Learning
- pg_plan_inspector
- pg_tuner

I graduated from graduate school in information engineering (M.S. in Information Engineering), have worked for several companies as a software developer and technical manager/director. I published seven books in the fields of database and system integration (4 PostgreSQL books and 3 MySQL books). In June 2019, the Chinese book of this document was published.

As a director of the Japan PostgreSQL Users Group (2010-2016), I organized the largest (non-commercial) technical seminar/lecture on PostgreSQL in Japan for more than six years, and also served as the program committee chair of the Japan PostgreSQL Conference in 2013 and as a member in 2008 and 2009.

In June 2022, my interview article was published in “PostgreSQL person of the week”.

Cuando era joven, vivió en Sudamérica por unos años. Recientemente, a veces vuelve a allí.

I am looking for a new job, applying ML and AI technologies to DBMS.

Blog

Contact

Please read the following FAQ before sending messages.

FAQ

After reading, send a message to my twitter in public.

or /

If you use email, please **provide at least two SNS addresses (e.g. LinkedIn, Twitter) for verification purposes**. Due to the XZ backdoor incident, I no longer accept contact from anonymous individuals.

Copyright

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:

Chapter Contents

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.

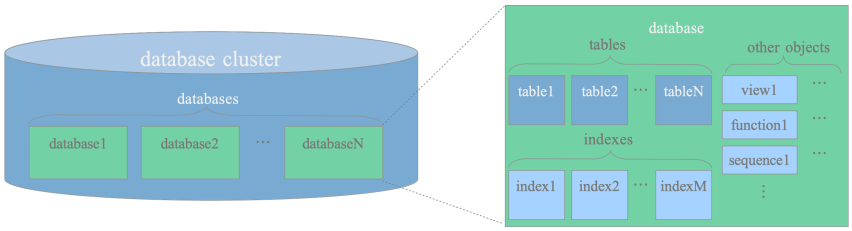

Figure 1.1. Logical structure of a database cluster.

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.

To find the desired OIDs, execute the following queries:

```
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 = 'sampletbl';
  relname  |  oid
-----------+-------
 sampletbl | 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.

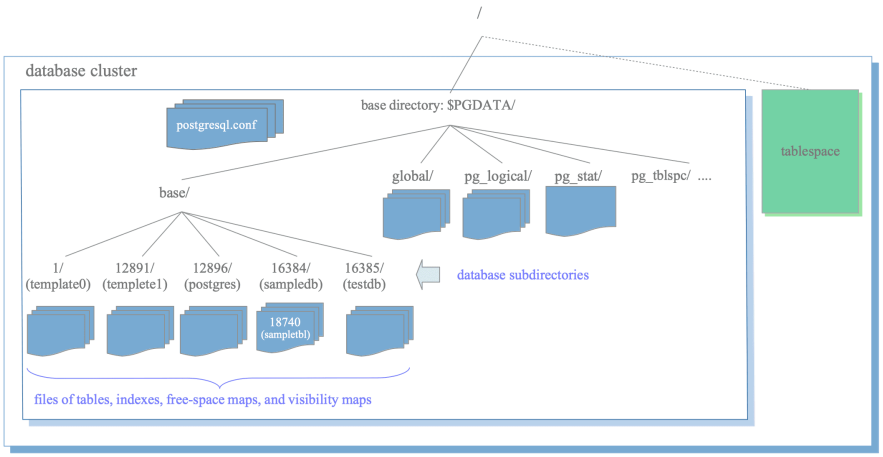

Figure 1.2. An example of database cluster.

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.
current_logfiles	A file recording the log file(s) currently written to by the logging collector.
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.

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. Replication Slots will be described in Section 11.4. (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.
pg_tblspc/	Subdirectory containing symbolic links to tablespaces.
pg_twophase/	Subdirectory containing state files for prepared transactions.
pg_wal/ (versions 10 or later)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed from *pg_xlog* in Version 10. WAL will be described in Chapter 9.
pg_xact/ (versions 10 or later)	Subdirectory containing transaction commit state data. It is renamed from *pg_clog* in Version 10. CLOG will be described in Section 5.4.
pg_xlog/ (versions 9.6 or earlier)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed to *pg_wal* 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, *relfilenode*. The relfilenode 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 relfilenode of the table 'sampletbl':

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

The oid and relfilenode values are equal in this case. Additionally, 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 relfilenode 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 relfilenode (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, relfilenode FROM pg_class WHERE relname = 'sampletbl';
  relname  |  oid  | relfilenode
```

```
-----------+-------+-------------
 sampletbl | 18740 |       18812
(1 row)
```

Info

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 relfilenode.1 and uses it. If the new file is filled up, PostgreSQL will create another new file named like relfilenode.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
```

Info

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

By examining the database subdirectories, it becomes apparent that each table has two associated files, suffixed with '_fsm' and '_vm'. These are referred to as 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.

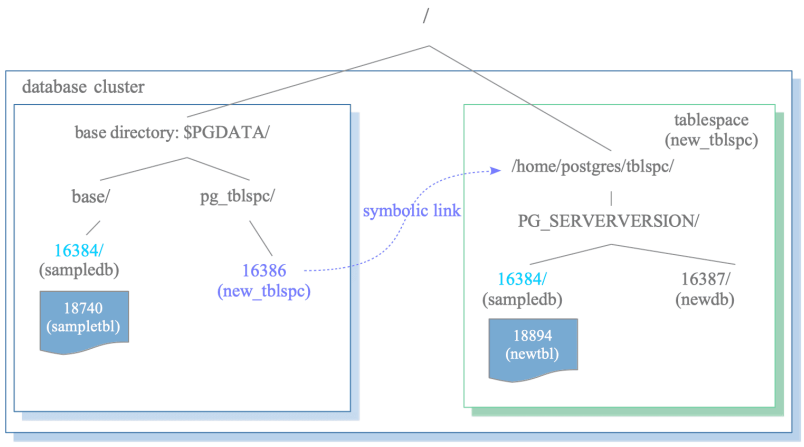

Figure 1.3. A Tablespace in the Database Cluster.

A tablespace is created within the directory specified during the execution of 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, creating a tablespace named '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
```

When a new database (OID 16387) is created within the tablespace, its directory is placed 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 a new table is created within a database located in the base directory, a new directory is first created under the version-specific subdirectory, named according to the OID of the database. The new table file is then placed in this newly 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.

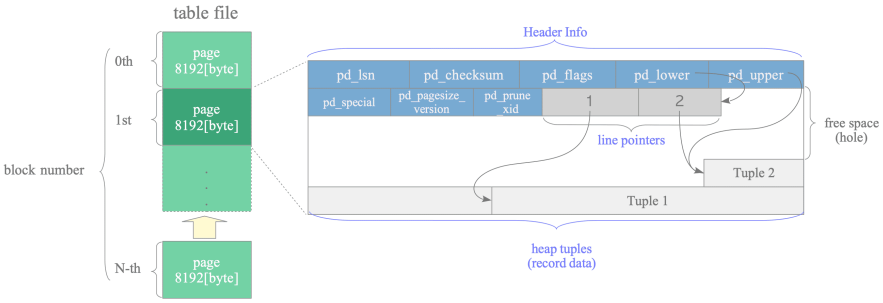

Figure 1.4. Page layout of a heap table file.

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:

   › PageHeaderData

   - *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 Section 9.1.2.
   - *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 *pd_tli*, which is 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.

Info

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

> **Info**
>
> 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 Figure 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 Figure 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 Section 9.4.

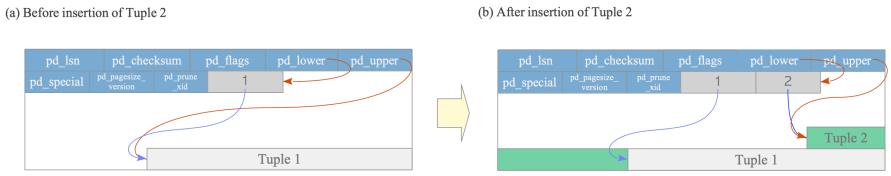

Figure 1.5. Writing of a heap tuple.

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 Figure 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[1], PostgreSQL reads the desired heap tuple using the obtained TID value.
  For example, in Figure 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.

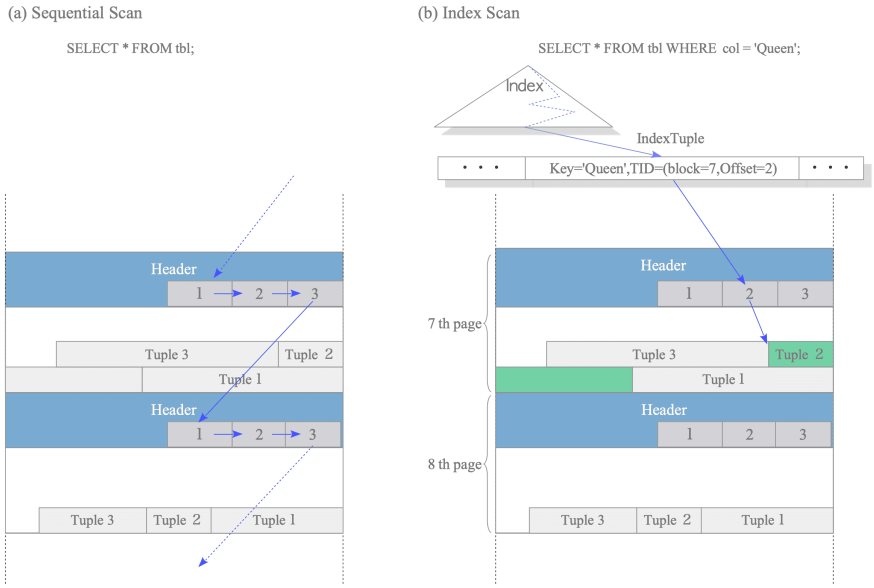

Figure 1.6. Sequential scan and index scan.

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 (GIN)

Info

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) | AAAAAAAAA
(1 row)
```

Index-Only-Scan will be described in details in Section 7.2.

---

1. 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. ↩

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.

Chapter Contents

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 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.

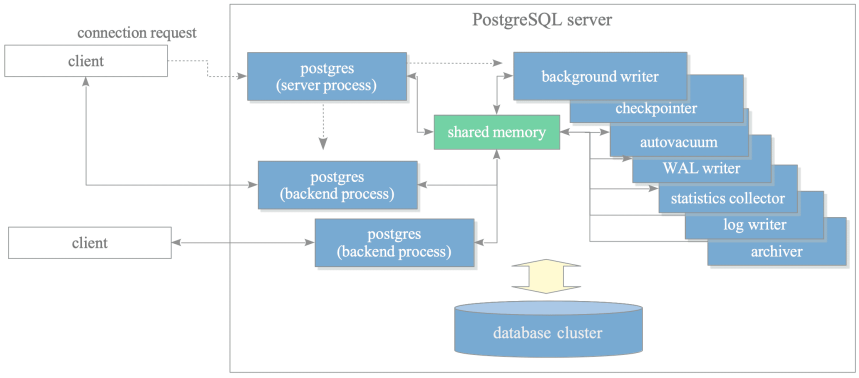

Figure 2.1. An example of the process architecture in PostgreSQL.

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 mentioned earlier, the postgres server process is the parent of all processes within a PostgreSQL server. In earlier versions, it was referred to as *postmaster*.

When the pg_ctl utility is executed with the *start* option, the postgres server process starts.

It then allocates a shared memory area, initiates various background processes, starts replication-related processes and background worker processes as needed, and waits for connection requests from clients. When a connection request is received, it spawns a backend process to handle the client session.

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.

Each backend process can only operate on one database at a time, requiring explicit database selection during connection.

PostgreSQL supports concurrent connections from multiple clients, with the maximum number controlled by the max_connections configuration parameter (default: 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. (In versions 9.1 or earlier, it was also responsible for the checkpoint process.)	Section 8.6
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 9.6.1
WAL Summarizer	This process tracks changes to all database blocks and writes these modifications to WAL summary files. Introduced in version 17.	Section 9.6.2
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

Info

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.

```
$ 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.

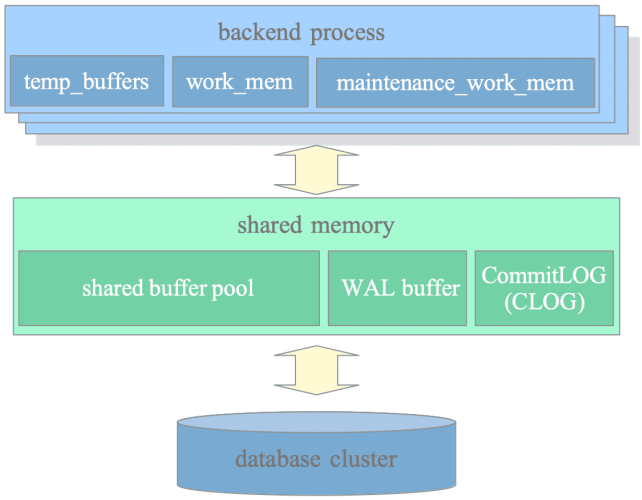

Figure 2.2. Memory architecture in PostgreSQL.

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
work_mem	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
maintenance_work_mem	Some kinds of maintenance operations (e.g., VACUUM, REINDEX) use this area.	Section 6.1
temp_buffers	The executor uses this area for storing temporary tables.	

In addition, Dynamic Shared Memory (DSM) was implemented in version 9.4 for Parallel Query. It is an on-demand memory space (i.e., it can be allocated as needed and released when no longer required) used for inter-process communication between PostgreSQL processes.

Dynamic Shared Memory (DSM)

DSM is an independent feature that can be used by users to send and receive data between PostgreSQL processes when developing extension modules. For example, Logical replication relies on DSM. the pg_prewarm contribution module also utilizes DSM.

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.

---

3. QUERY PROCESSING

As 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 four 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.
- **Part 4:** Sections 3.7.
  This part briefly explains Parallel Query.

PostgreSQL supports two technically interesting and practical features: Foreign Data Wrappers (FDW) and JIT compilation which is supported from version 11. The FDW will be described in Chapter 4. The JIT compilation is out of scope of this document.

Chapter Contents

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.

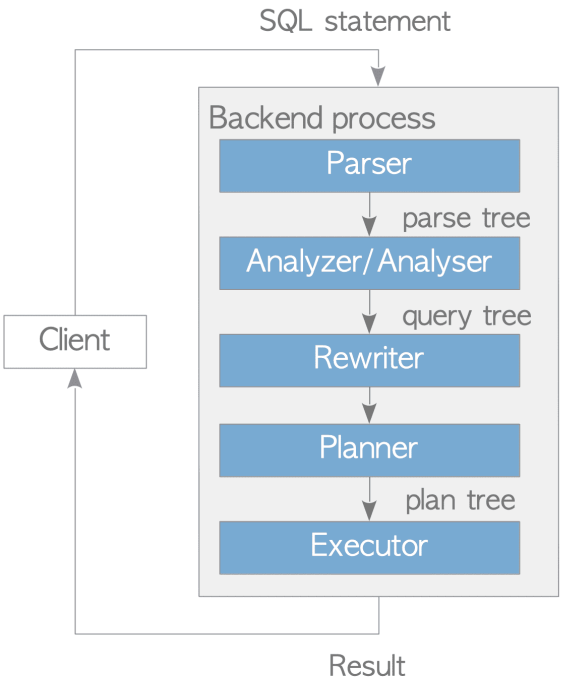

Figure 3.1. Query Processing.

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.

Info

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 Figure 3.2(a).

SelectStmt

```
typedef struct SelectStmt
{
        NodeTag         type;

        /*
         * These fields are used only in "leaf" SelectStmts.
         */
        List       *distinctClause;     /* NULL, list of DISTINCT ON exprs, or
                                         * lcons(NIL,NIL) for all (SELECT DISTINCT) */
        IntoClause *intoClause;         /* target for SELECT INTO */
        List       *targetList;         /* the target list (of ResTarget) */
        List       *fromClause;         /* the FROM clause */
        Node       *whereClause;        /* WHERE qualification */
        List       *groupClause;        /* GROUP BY clauses */
        Node       *havingClause;       /* HAVING conditional-expression */
        List       *windowClause;       /* WINDOW window_name AS (...), ... */

        /*
         * In a "leaf" node representing a VALUES list, the above fields are all
         * null, and instead this field is set.  Note that the elements of the
         * sublists are just expressions, without ResTarget decoration. Also note
         * that a list element can be DEFAULT (represented as a SetToDefault
         * node), regardless of the context of the VALUES list. It's up to parse
         * analysis to reject that where not valid.
         */
        List       *valuesLists;        /* untransformed list of expression lists */

        /*
         * These fields are used in both "leaf" SelectStmts and upper-level
         * SelectStmts.
         */
        List       *sortClause;         /* sort clause (a list of SortBy's) */
        Node       *limitOffset;        /* # of result tuples to skip */
        Node       *limitCount;         /* # of result tuples to return */
        List       *lockingClause;      /* FOR UPDATE (list of LockingClause's) */
        WithClause *withClause;         /* WITH clause */

        /*
         * These fields are used only in upper-level SelectStmts.
         */
        SetOperation op;                /* type of set op */
        bool            all;            /* ALL specified? */
        struct SelectStmt *larg;        /* left child */
        struct SelectStmt *rarg;        /* right child */
        /* Eventually add fields for CORRESPONDING spec here */
} SelectStmt;
```

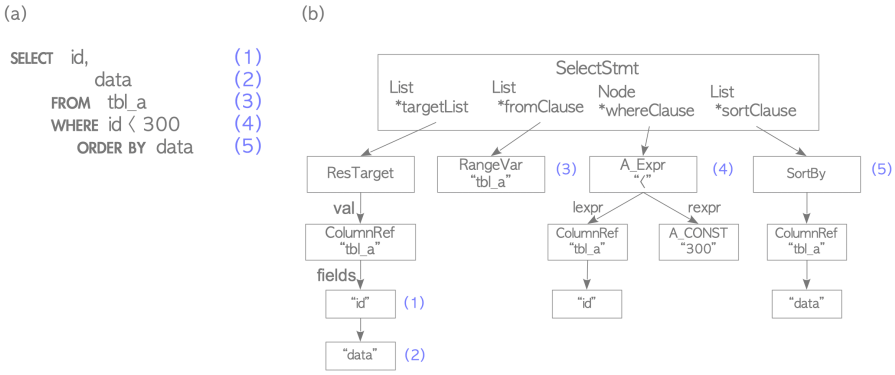

Figure 3.2. An example of a parse tree.

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.

Query

```
/*
 * Query -
 *    Parse analysis turns all statements into a Query tree
 *    for further processing by the rewriter and planner.
 *
 *    Utility statements (i.e. non-optimizable statements) have the
 *    utilityStmt field set, and the Query itself is mostly dummy.
 *    DECLARE CURSOR is a special case: it is represented like a SELECT,
 *    but the original DeclareCursorStmt is stored in utilityStmt.
 *
 *    Planning converts a Query tree into a Plan tree headed by a PlannedStmt
 *    node --- the Query structure is not used by the executor.
 */
typedef struct Query
{
    NodeTag     type;

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

    /* where did I come from? */
    QuerySource querySource pg_node_attr(query_jumble_ignore);

    /*
     * query identifier (can be set by plugins); ignored for equal, as it
     * might not be set; also not stored.  This is the result of the query
     * jumble, hence ignored.
     */
    uint64      queryId pg_node_attr(equal_ignore, query_jumble_ignore, read_write_ignore,
read_as(0));

    /* do I set the command result tag? */
    bool        canSetTag pg_node_attr(query_jumble_ignore);

    Node       *utilityStmt;    /* non-null if commandType == CMD_UTILITY */

    /*
     * rtable index of target relation for INSERT/UPDATE/DELETE/MERGE; 0 for
     * SELECT.  This is ignored in the query jumble as unrelated to the
     * compilation of the query ID.
     */
    int         resultRelation pg_node_attr(query_jumble_ignore);

    /* has aggregates in tlist or havingQual */
    bool        hasAggs pg_node_attr(query_jumble_ignore);
    /* has window functions in tlist */
    bool        hasWindowFuncs pg_node_attr(query_jumble_ignore);
    /* has set-returning functions in tlist */
    bool        hasTargetSRFs pg_node_attr(query_jumble_ignore);
    /* has subquery SubLink */
    bool        hasSubLinks pg_node_attr(query_jumble_ignore);
    /* distinctClause is from DISTINCT ON */
    bool        hasDistinctOn pg_node_attr(query_jumble_ignore);
    /* WITH RECURSIVE was specified */
    bool        hasRecursive pg_node_attr(query_jumble_ignore);
    /* has INSERT/UPDATE/DELETE in WITH */
    bool        hasModifyingCTE pg_node_attr(query_jumble_ignore);
    /* FOR [KEY] UPDATE/SHARE was specified */
    bool        hasForUpdate pg_node_attr(query_jumble_ignore);
    /* rewriter has applied some RLS policy */
    bool        hasRowSecurity pg_node_attr(query_jumble_ignore);
    /* is a RETURN statement */
    bool        isReturn pg_node_attr(query_jumble_ignore);

    List       *cteList;        /* WITH list (of CommonTableExpr's) */

    List       *rtable;         /* list of range table entries */

    /*
     * list of RTEPermissionInfo nodes for the rtable entries having
     * perminfoindex > 0
     */
    List       *rteperminfos pg_node_attr(query_jumble_ignore);
    FromExpr   *jointree;       /* table join tree (FROM and WHERE clauses);
                                 * also USING clause for MERGE */

    List       *mergeActionList;    /* list of actions for MERGE (only) */
    /* whether to use outer join */
    bool        mergeUseOuterJoin pg_node_attr(query_jumble_ignore);

    List       *targetList;     /* target list (of TargetEntry) */

    /* OVERRIDING clause */
    OverridingKind override pg_node_attr(query_jumble_ignore);

    OnConflictExpr *onConflict; /* ON CONFLICT DO [NOTHING | UPDATE] */

    List       *returningList;  /* return-values list (of TargetEntry) */

    List       *groupClause;    /* a list of SortGroupClause's */
    bool        groupDistinct;  /* is the group by clause distinct? */

    List       *groupingSets;   /* a list of GroupingSet's if present */

    Node       *havingQual;     /* qualifications applied to groups */

    List       *windowClause;   /* a list of WindowClause's */
```

```
    List       *distinctClause; /* a list of SortGroupClause's */

    List       *sortClause;     /* a list of SortGroupClause's */

    Node       *limitOffset;    /* # of result tuples to skip (int8 expr) */
    Node       *limitCount;     /* # of result tuples to return (int8 expr) */
    LimitOption limitOption;    /* limit type */

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

    Node       *setOperations;  /* set-operation tree if this is top level of
                                 * a UNION/INTERSECT/EXCEPT query */

    /*
     * A list of pg_constraint OIDs that the query depends on to be
     * semantically valid
     */
    List       *constraintDeps pg_node_attr(query_jumble_ignore);

    /* a list of WithCheckOption's (added during rewrite) */
    List       *withCheckOptions pg_node_attr(query_jumble_ignore);

    /*
     * The following two fields identify the portion of the source text string
     * containing this query.  They are typically only populated in top-level
     * Queries, not in sub-queries.  When not set, they might both be zero, or
     * both be -1 meaning "unknown".
     */
    /* start location, or -1 if unknown */
    int         stmt_location;
    /* length in bytes; 0 means "rest of string" */
    int         stmt_len pg_node_attr(query_jumble_ignore);
} Query;
```

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

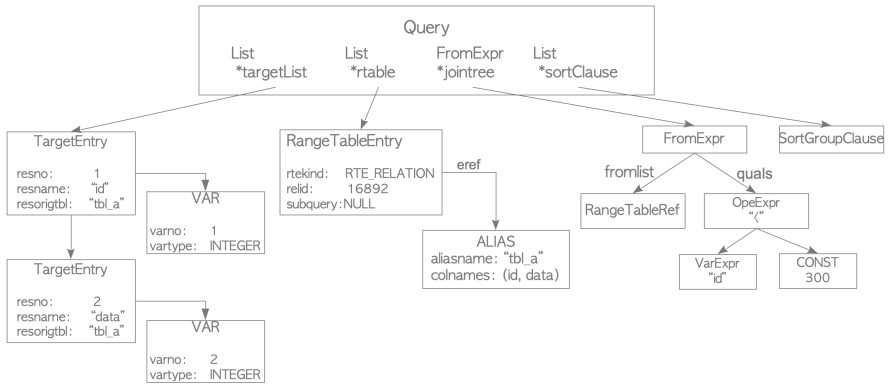

Figure 3.3. An example of a query tree.

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 Figure 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*.

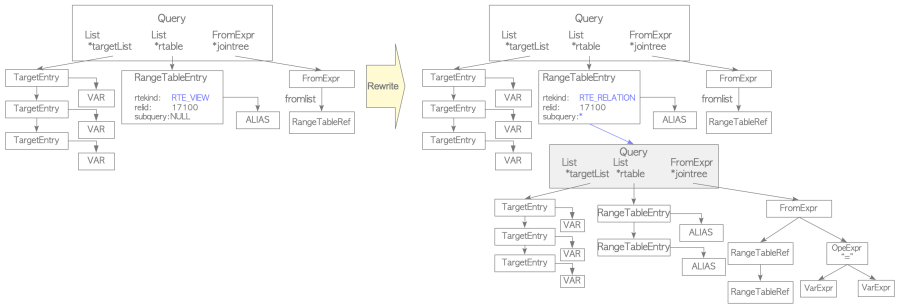

Figure 3.4. An example of the rewriter stage.

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. To use hints in queries, consider installing the pg_hint_plan extension.

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 Figure 3.5.

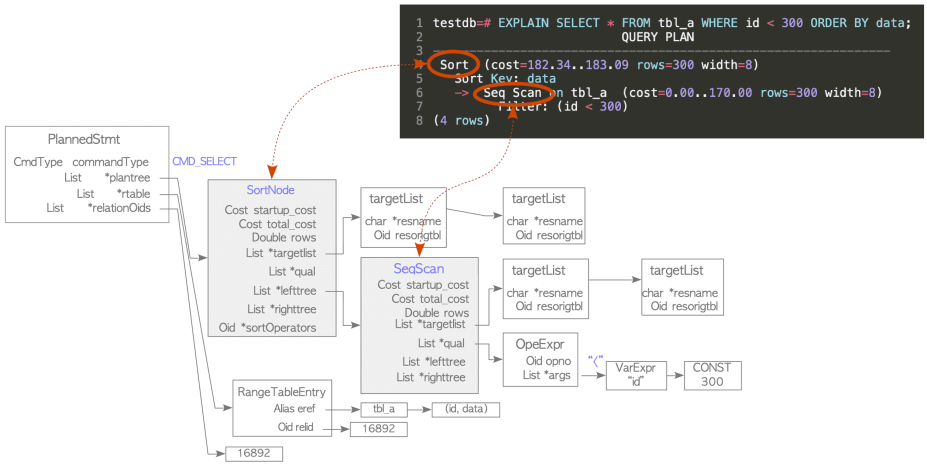

Figure 3.5. A simple plan tree and the relationship between the plan tree and the result of the EXPLAIN command.

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 Figure 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.

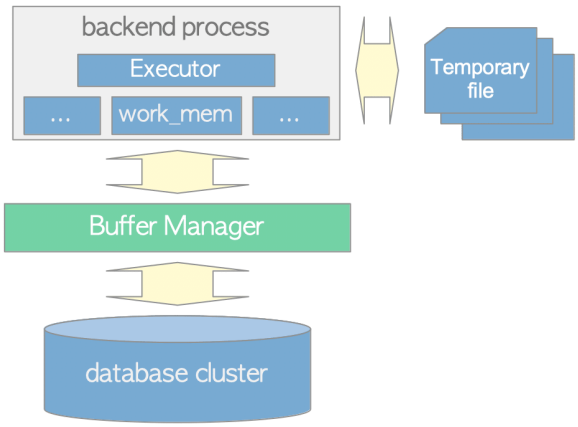

Figure 3.6. The relationship among the executor, buffer manager and temporary files.

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 '$\text{Filter : (id} <= 8000)$' 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.

Info

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)
```

$$N_{index,tuple} = 10000 \tag{3}$$
$$N_{index,page} = 30 \tag{4}$$

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:

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

where H_{index} is the height of the index tree. The detail of this calculation is explained in the comments of btcostestimate().

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

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

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:

$$\text{'run cost'} = (\text{'index cpu cost'} + \text{'table cpu cost'}) + (\text{'index IO cost'} + \text{'table IO cost'}).$$

Info

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:

$$\begin{aligned}
\text{'index cpu cost'} &= \text{Selectivity} \times N_{index,tuple} \times (\text{cpu_index_tuple_cost} + \text{qual_op_cost}) \\
\text{'table cpu cost'} &= \text{Selectivity} \times N_{tuple} \times \text{cpu_tuple_cost} \\
\text{'index IO cost'} &= \text{ceil}(\text{Selectivity} \times N_{index,page}) \times \text{random_page_cost}
\end{aligned}$$

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.
  $(\text{Selectivity} \times N_{tuple})$ means the number of the table tuples to be read;
  $(\text{Selectivity} \times N_{index,page})$ 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.

› countries

```
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.

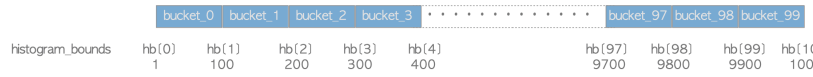

Figure 3.7. Buckets and histogram_bounds.

Next, the calculation of the selectivity will be shown using the example in this subsection. The query has a WHERE clause '$\mathbf{data} <= 240$' 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:

$$\begin{aligned}\text{Selectivity} &= \frac{2 + (240 - \text{hb}[2])/(\text{hb}[3] - \text{hb}[2])}{100} \\ &= \frac{2 + (240 - 200)/(300 - 200)}{100} = \frac{2 + 40/100}{100} \\ &= 0.024\end{aligned} \tag{6}$$

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

$$\text{'index cpu cost'} = 0.024 \times 10000 \times (0.005 + 0.0025) = 1.8 \tag{7}$$
$$\text{'table cpu cost'} = 0.024 \times 10000 \times 0.01 = 2.4 \tag{8}$$
$$\text{'index IO cost'} = \text{ceil}(0.024 \times 30) \times 4.0 = 4.0 \tag{9}$$

$\mathbf{\text{'table IO cost'}}$ is defined by the following equation:

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

$\mathbf{\text{max\_IO\_cost}}$ 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:

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

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

$$\text{max\_IO\_cost} = 45 \times 4.0 = 180.0 \tag{10}$$

$\mathbf{\text{min\_IO\_cost}}$ 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:

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

In this case,

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

$\mathbf{\text{indexCorrelation}}$ is described in detail in ℹ below, and in this example,

$$\text{indexCorrelation} = 1.0 \tag{12}$$

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

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

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

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

ℹ 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 Figure 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 Figure 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.

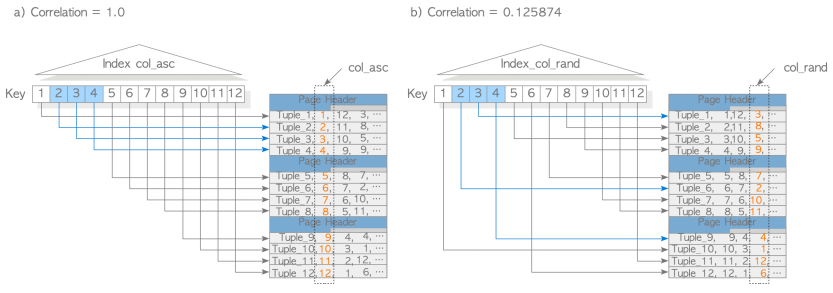

Figure 3.8. Index correlation.

3.2.2.3. Total Cost

According to (3) and (14),

$$\text{'total cost'} = 0.285 + 13.2 = 13.485 \tag{15}$$

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 '$\mathbf{IndexCond}: (\text{data} <= 240)$' 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.

Index Condition

According to this post, EXPLAIN command in PostgreSQL does not distinguish between the access predicate and index filter predicate. Therefore, when analyzing the output of EXPLAIN, it is important to consider both the index conditions and the estimated value of rows.

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 $\textbf{start-up cost}$ 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 $\textbf{run cost}$ 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.

Path

```
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

```
/*----------
 * 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_array_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.
     */
    /* set of all result relids */
    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);

    /* 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_cxt 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_aggrefs().
     */
    /* 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 Figure 3.9(b).

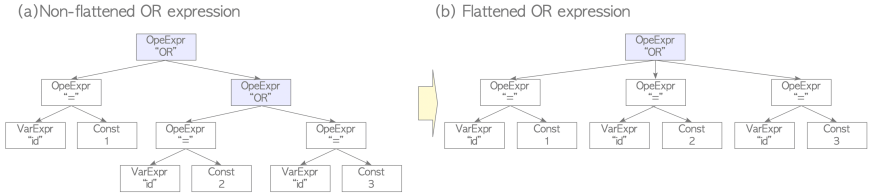

Figure 3.9. An example of flattening AND/OR expressions.

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 Figure 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.

RelOptInfo

```
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  *eclass_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_eclass_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.

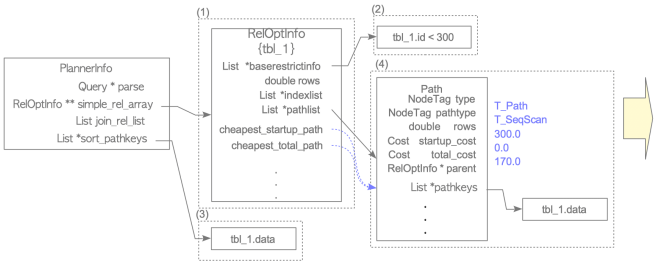

Figure 3.10. How to get the cheapest path of Example 1

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.

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.

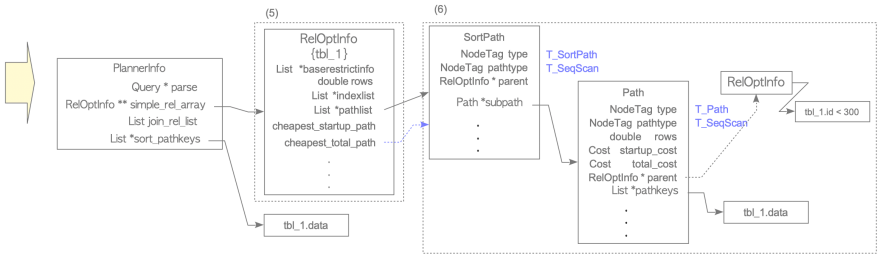

Figure 3.11. How to get the cheapest path of Example 1. (continued from Figure 3.10)

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.

▾ SortPath

```
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.

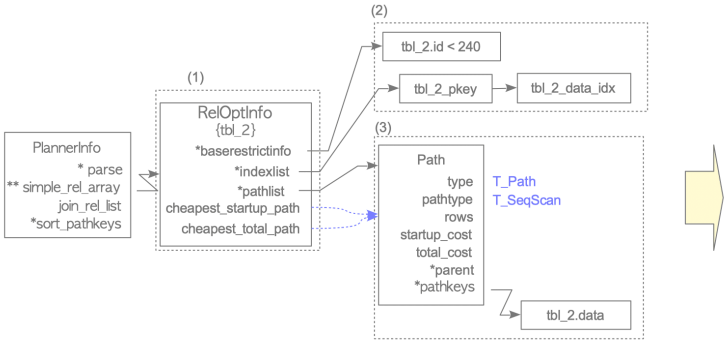

Figure 3.12. How to get the cheapest path of Example 2.

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 '$\mathbf{id} < 240$' 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.

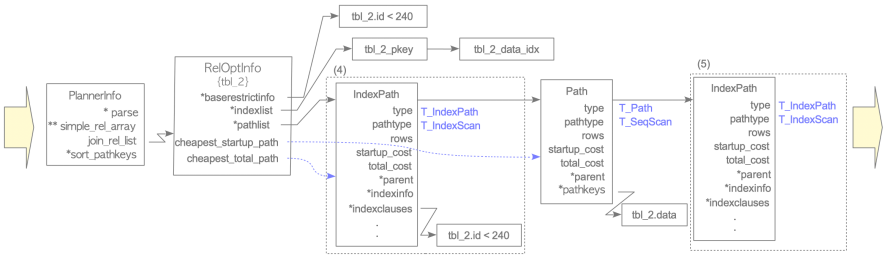

Figure 3.13. How to get the cheapest path of Example 2. (continued from Figure 3.12)

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.

▾ IndexPath

```
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. They do
 *      not contain any information about included attributes.
 *
 *      sortopfamily[], reverse_sort[], and nulls_first[] have
 *      nkeycolumns entries, if the index is ordered; but if it is unordered,
 *      those pointers are NULL.
 *
 *      Zeroes in the indexkeys[] array indicate index columns that are
 *      expressions; there is one element in indexprs for each such column.
 *
 *      For an ordered index, reverse_sort[] and nulls_first[] describe the
 *      sort ordering of a forward indexscan; we can also consider a backward
 *      indexscan, which will generate the reverse ordering.
 *
 *      The indexprs and indpred expressions have been run through
 *      prepqual.c and eval_const_expressions() for ease of matching to
 *      WHERE clauses. indpred is in implicit-AND form.
 *
 *      indextlist is a TargetEntry list representing the index columns.
 *      It provides an equivalent base-relation Var for each simple column,
 *      and links to the matching indexprs element for each expression column.
 *
 *      While most of these fields are filled when the IndexOptInfo is created
 *      (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(nkeycolumns));
    /* 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       *indexprs 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        amhasgetbitmap;
    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**
>
> 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.

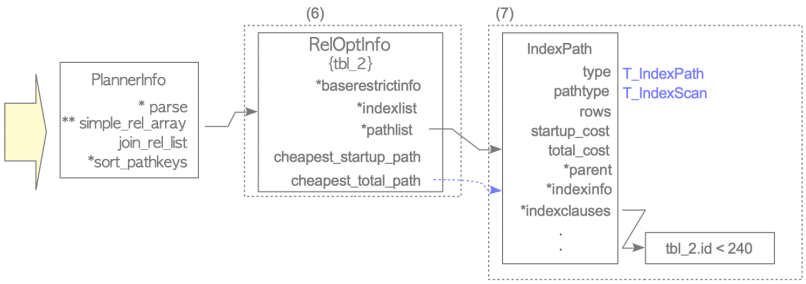

Figure 3.14. How to get the cheapest path of Example 2. (continued from Figure 3.13)

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.

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.

PlannedStmt

```
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 Query) */

    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 one */

    /* 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       *invalItems;     /* 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.

PlanNode

```
/* ----------------
 *      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;
```

ScanNode

```
/*
 * ==========
 * 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 */
} 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 Figure 3.15(a).

SortNode

```
/* ----------------
 *      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;
```

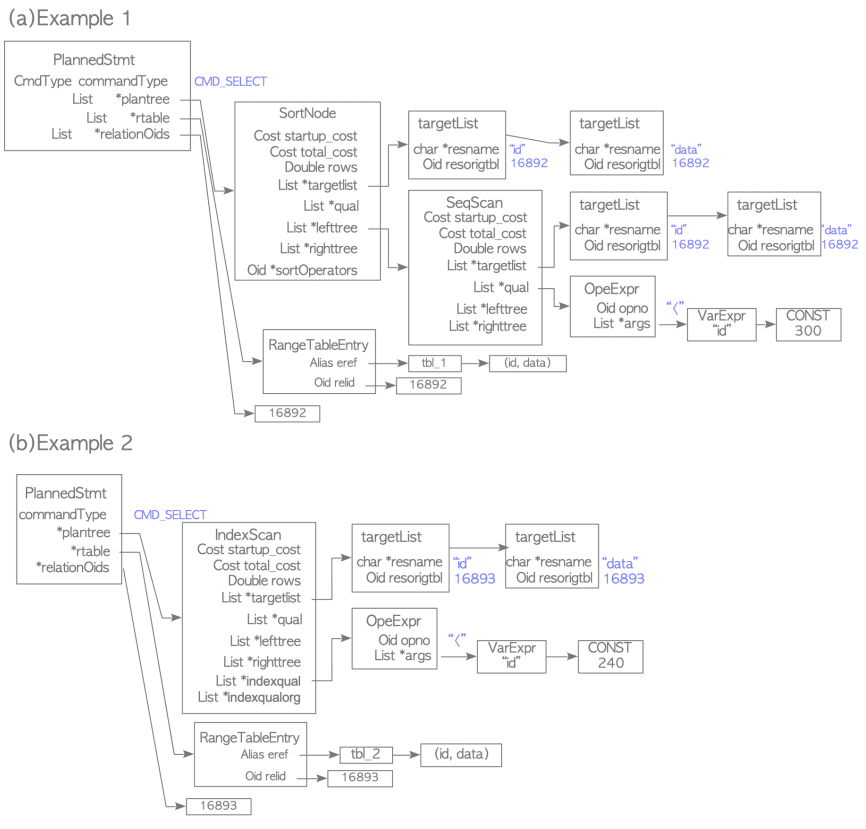

Figure 3.15. Examples of plan trees.

In the SortNode, the lefttree points to the SeqScanNode.

In the SeqScanNode, the qual holds the WHERE clause '$\mathrm{id} < 300$'.

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 Figure 3.14 is the index scan path, so the plan tree is composed of an `IndexScanNode` structure alone. See Figure 3.15(b).

▾ IndexScanNode

```
/* ----------------
 *      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 table */
} 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 '$\mathrm{id} < 240$' is an access predicate, so it is stored in the indexqual of the IndexScanNode.

3.4. EXECUTOR PERFORMANCE

This section explains how the executor processes queries and how aggregate functions are handled.

3.4.1. 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 (SeqScan) 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.

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.

```
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**: The SeqScan node, defined in nodeSeqscan.c, sequentially scans the target table and sends tuples to the SortNode.
- **Line 4**: The SortNode receives tuples one by one from the SeqScan node, stores them in the work_mem buffer, and sorts the tuples after the SeqScan is complete.

Figure 3.16 illustrates how the sort processing works.

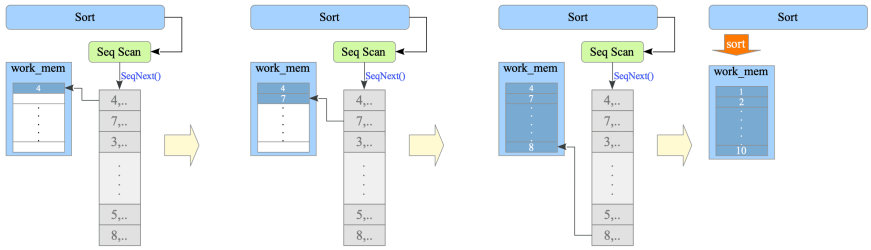

Figure 3.16. Sort Processing in the Executor.

From the Executor's perspective, it starts a SortNode, runs a SeqScan within the SortNode to obtain tuples sequentially, and then sorts these tuples in memory (work_mem).

Scanning Functions

As mentioned in Section 1.3, data tuples are stored in 8KB physical blocks. To access a single data tuple, the Executor must traverse many layers, such as Buffer Manager (see Chapter 8) and Concurrency Control (see Chapter 5). Furthermore, the access method differs depending on whether the data block belongs to a table or an index.

The method of accessing a single data tuple by the Executor is highly abstracted, and the complexity of the intermediate layers is hidden.

For example, in the case of a sequential scan, by repeatedly calling the SeqNext() function, rows in a table can be accessed sequentially, taking transactions into account. For an index scan, an index scan can be performed by repeatedly calling the IndexNext() function.

3.4.1.1. 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 ----------------------
 5  Sort  (cost=3944070.01..3945895.01 rows=730000 width=4104) (actual time=885.648..1033.746
 6 rows=730000 loops=1)
 7    Sort Key: id
 8    Sort Method: external sort  Disk: 10000kB
 9    ->  Seq Scan on tbl_25m  (cost=0.00..10531.00 rows=730000 width=4104) (actual time=0.024..102.548
10 rows=730000 loops=1)
    Planning time: 1.548 ms
    Execution time: 1109.571 ms
   (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} . {sequencial 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/base/pgsql_tmp*
-rw-------  1 postgres  postgres  10240000 12  4 14:18 pgsql_tmp8903.5
```

3.4.2. Aggregate Functions

In practice, calculating aggregate functions is also an important role of the database systems. We briefly explore how aggregate functions are handled, using sum, average and variance.

In the following, we use the following table `d`:

```
testdb=# CREATE TABLE d (x DOUBLE PRECISION);
CREATE TABLE
testdb=# INSERT INTO d SELECT GENERATE_SERIES(1, 10);
INSERT 0 10
```

For example, when the sum function is issued, the following plan tree (Figure 3.17) is created.

```
testdb=# SELECT sum(x) FROM d;
 sum
-----
  55
(1 row)
```

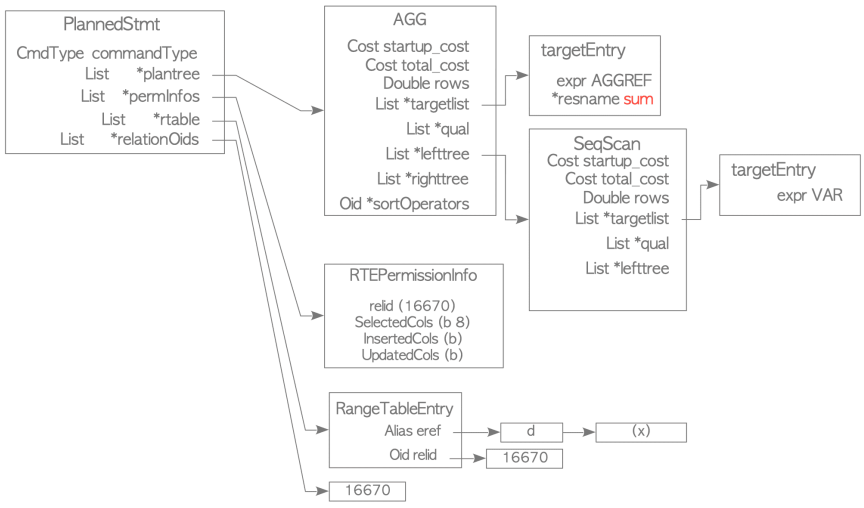

Figure 3.17. A Plan Tree for the Sum Function.

The result of the EXPLAIN command is shown below:

```
1 testdb=# EXPLAIN SELECT sum(x) FROM d;
2                         QUERY PLAN
```

```
3 -------------------------------------------------------
4  Aggregate  (cost=38.25..38.26 rows=1 width=8)
5    ->  Seq Scan on d  (cost=0.00..32.60 rows=2260 width=8)
6 (2 rows)
```

- **Line 5**: The SeqScan node sequentially scans the target table and sends the value of the target column (datum) to the AggregateNode.
- **Line 4**: The AggregateNode receives the datum from the SeqScan node and processes it according to the specified aggregate function (e.g., sum, average, variance, count, etc.).

Figure 3.18 orverviews how the aggregate function is processed.

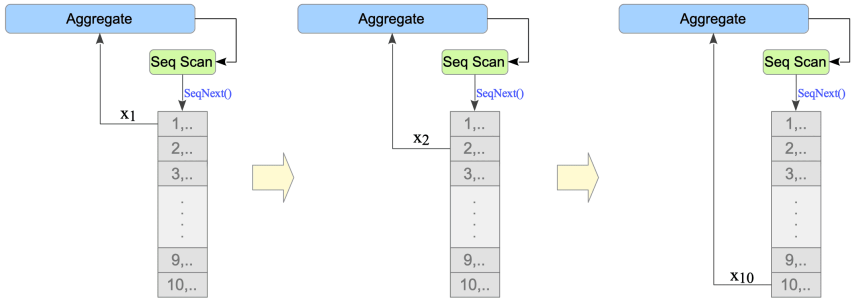

Figure 3.18. Aggregate Function Processing in the Executor.

In the following subsections, we will explore how aggregate functions are processed using concrete examples.

3.4.2.1. Sum and Average

The formulas for the sum S_n and average A_n are given by:

$$S_n = \sum_{i=1}^{n} x_i \tag{16}$$

$$A_n = \frac{1}{n} \sum_{i=1}^{n} x_i = \frac{1}{n} S_n \tag{17}$$

where x_i is the value of the target column at index i.

As is evident, the only difference between the sum and average is whether or not we divide by the number of elements, n.

Internally, an `Aggregate` node accumulates the scanned values sequentially. When returning the result, it returns the sum for the sum operation, and the sum divided by n for the average.

Following is the pseudocode.

```
d =[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

N = 0
S = 0

# Seq Scan
for x in d:

    # Aggregate: Sum
    N += 1
    S += x

print("Sum=", S)
print("Avg=", S/N)
```

3.4.2.2. Variance

For simplicity, we define variance as follows:

$$V_n = \sum_{i=1}^{n} (x_i - A_n)^2 \tag{18}$$

From this definition, we can define:

- Sample Variance (var_samp): $\frac{1}{n-1} V_n$ (used when data represents a sample)
- Population Variance (var_pop): $\frac{1}{n} V_n$ (used when data represents the entire population)

```
testdb=# SELECT var_samp(x) FROM d;
     var_samp
------------------
 9.166666666666666
(1 row)
```

```
testdb=# SELECT var_pop(x) FROM d;
 var_pop
---------
    8.25
(1 row)
```

One-Pass Variance Calculation (Versions 11 or earlier)

A naive implementation of the variance formula (18) requires two passes over the data: one to calculate the average and another to calculate the squared differences.

However, we can optimize this to a single-pass calculation using the following formula (For details, refer to the following ℹ):

$$V_n = \sum_{i=1}^{n} x_i^2 - \frac{1}{n}(S_n)^2 \tag{19}$$

This allows us to calculate the variance in a single pass by iteratively calculating x_i^2 and S_i.

Following is the pseudocode.

```
d =[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

N = 0
S = 0
X2 = 0

# SeqScan
for x in d:

    # Aggregate: Variance
    N += 1
    S += x
    X2 += x**2

V = X2 - S**2/N

print("V=", V)
print("Var_samp=", V/(N-1))
print("Var_pop=", V/N)
```

ℹ Rewriting from 2-Pass to 1-Pass Method

$$\begin{aligned}
V_n &= \sum_{i=1}^{n}(x_i - A_n)^2 = \sum_{i=1}^{n}(x_i^2 - 2A_n x_i + (A_n)^2) = \sum_{i=1}^{n} x_i^2 - 2A_n \sum_{i=1}^{n} x_i + \sum_{i=1}^{n}(A_n)^2 \\
&= \sum_{i=1}^{n} x_i^2 - 2\frac{S_n}{n}\sum_{i=1}^{n} x_i + \sum_{i=1}^{n}\left(\frac{S_n}{n}\right)^2 = \sum_{i=1}^{n} x_i^2 - 2\frac{S_n}{n}S_n + n\cdot\left(\frac{S_n}{n}\right)^2 \\
&= \sum_{i=1}^{n} x_i^2 - \frac{2}{n}(S_n)^2 + \frac{1}{n}(S_n)^2 \\
&= \sum_{i=1}^{n} x_i^2 - \frac{1}{n}(S_n)^2
\end{aligned}$$

Youngs and Cramer method (Versions 12 or later)

A paper titled "A Closer Look at Variance ImplementationsIn Modern Database Systems" pointed out the low accuracy of PostgreSQL's variance function in 2016.

To address this issue, a patch was created in November 2018, and the Youngs and Cramer method was subsequently adopted in version 12. This method offers a more accurate way to calculate variance.

$$\begin{cases} V_1 &= 0 \\ V_n &= V_{n-1} + \frac{1}{n(n-1)}(nx_n - S_n)^2 \end{cases} \tag{20}$$

Following is the pseudocode of Youngs and Cramer method:

```
d =[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

N = 0
S = 0
V = 0

# SeqScan
for x in d:

    # Aggregate: Variance
    N += 1
    S += x
    if 1 < N:
        V += (x * N - S)**2 / (N * (N-1))

print("V=", V)
print("Var_samp=", V/(N-1))
print("Var_pop=", V/N)
```

ℹ Derivation of the Youngs and Cramer Method

The Youngs and Cramer method is an improvement on the Welfold method. To derive the Youngs and Cramer method, we first examine the Welfold method.

Welfold method:

The Welfold method for calculating variance is defined as follows:

$$V_n = V_{n-1} + \frac{n-1}{n}(x_n - A_{n-1})^2$$

This can be rewritten as a recurrence relation:

$$\begin{aligned}
V_n &= \sum_{i=1}^{n}(x_i - A_n)^2 = \sum_{i=1}^{n-1}(x_i - A_n)^2 + (x_n - A_n)^2 \\
&= \sum_{i=1}^{n-1}\left((x_i - \frac{1}{n}((n-1)A_{n-1} + x_n)\right)^2 + \left((x_n - \frac{1}{n}((n-1)A_{n-1} + x_n)\right)^2 \\
&= \sum_{i=1}^{n-1}\left((x_i - A_{n-1}) - \frac{1}{n}(x_n - A_{n-1})\right)^2 + \left(\frac{1}{n}(nx_n - (n-1)A_{n-1} - x_n)\right)^2 \\
&= \sum_{i=1}^{n-1}\left((x_i - A_{n-1})^2 - \frac{2(x_n - A_{n-1})}{n}(x_i - A_{n-1}) + \frac{1}{n^2}(x_n - A_{n-1})^2\right) \\
&\quad + \left(\frac{n-1}{n}\right)^2(x_n - A_{n-1})^2 \\
&= \sum_{i=1}^{n-1}(x_i - A_{n-1})^2 - \frac{2(x_n - A_{n-1})}{n}\sum_{i=1}^{n-1}(x_i - A_{n-1}) + \frac{1}{n^2}\sum_{i=1}^{n-1}(x_n - A_{n-1})^2 \\
&\quad + \left(\frac{n-1}{n}\right)^2(x_n - A_{n-1})^2 \\
&= V_{n-1} - \frac{2(x_n - A_{n-1})}{n}\left(\sum_{i=1}^{n-1}x_i - (n-1)A_{n-1}\right) + \frac{1}{n^2}(n-1)\cdot(x_n - A_{n-1})^2 \\
&\quad + \left(\frac{n-1}{n}\right)^2(x_n - A_{n-1})^2 \\
&= V_{n-1} - \frac{2(x_n - A_{n-1})}{n}(S_{n-1} - S_{n-1}) + \frac{n-1}{n^2}(x_n - A_{n-1})^2 + \left(\frac{n-1}{n}\right)^2(x_n - A_{n-1})^2 \\
&= V_{n-1} - \frac{2(x_n - A_{n-1})}{n}\cdot 0 + \left(\frac{n-1}{n^2} + \left(\frac{n-1}{n}\right)^2\right)(x_n - A_{n-1})^2 \\
&= V_{n-1} + \frac{n-1}{n}(x_n - A_{n-1})^2
\end{aligned}$$

Youngs and Cramer method:

The Youngs and Cramer method is a modification of the Welfold method, where the average A_{n-1} is replaced by the sum S_{n-1} and further optimizations are made. The recurrence relation for the Youngs and Cramer method is:

$$\begin{aligned}
V_n &= V_{n-1} + \frac{n-1}{n}\left(x_n - \frac{S_{n-1}}{n-1}\right)^2 = V_{n-1} + \frac{n-1}{n}\left(\frac{(n-1)x_n - S_{n-1}}{n-1}\right)^2 \\
&= V_{n-1} + \frac{1}{n(n-1)}((n-1)x_n - S_{n-1})^2 = V_{n-1} + \frac{1}{n(n-1)}(nx_n - x_n - (S_n - x_n))^2 \\
&= V_{n-1} + \frac{1}{n(n-1)}(nx_n - S_n)^2
\end{aligned}$$

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 the references:

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

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.

Section Contents

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.

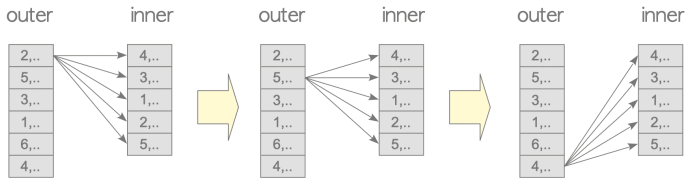

Figure. 3.19. Nested loop join.

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 ℹ 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 3.20 illustrates how the materialized nested loop join performs. Scanning materialized tuples is internally called **rescan**.

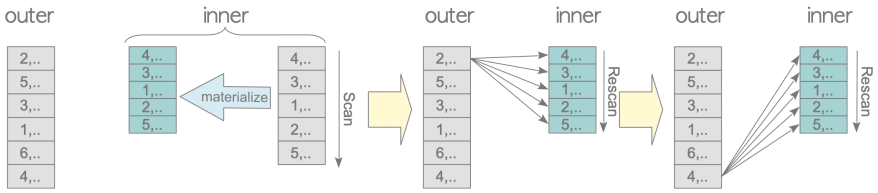

Figure 3.20. Materialized nested loop join.

ℹ 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.

```
1 testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id;
2                               QUERY PLAN
3 ------------------------------------------------------------------------
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:

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

where $C^{total}_{outer,seqscan}$ is the total scan cost of the outer table and $C^{total}_{materialize}$ 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 Figure 3.21. 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.

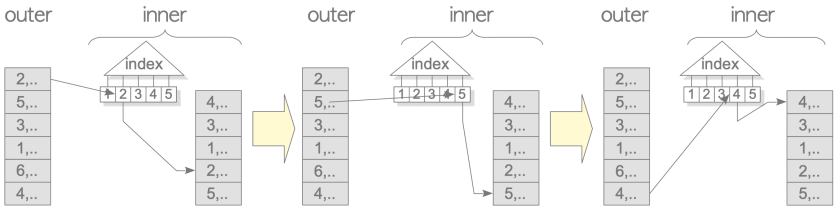

Figure 3.21. Indexed nested loop join.

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 '$(\mathrm{id} = \mathrm{b.id})$', which is shown in Line 7.

In the index condition '$(\mathrm{id} = \mathrm{b.id})$' in Line 7, '$\mathrm{b.id}$' 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:

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

where $C^{total}_{inner,parameterized}$ 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 as follows:

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

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

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 Figure 3.22.

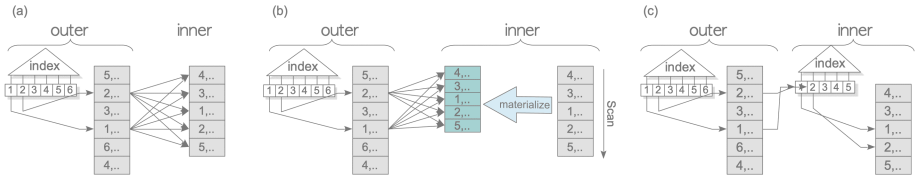

Figure 3.22. The three variations of the nested loop join with an outer index scan.

The results of these joins' EXPLAIN are shown below:

(a) Nested loop join with outer index scan

```
testdb=# SET enable_hashjoin TO off;
```

```
SET
testdb=# SET enable_mergejoin TO off;
SET
testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_b AS b WHERE c.id = b.id AND c.id = 500;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.29..93.81 rows=1 width=16)
   ->  Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..8.30 rows=1 width=8)
         Index Cond: (id = 500)
   ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=1 width=8)
         Filter: (id = 500)
(5 rows)
```

(2) Materialized nested loop join with outer index scan

```
testdb=# SET enable_hashjoin TO off;
SET
testdb=# SET enable_mergejoin TO off;
SET
testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_b AS b WHERE c.id = b.id AND c.id < 40 AND b.id < 10;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.29..99.76 rows=1 width=16)
   Join Filter: (c.id = b.id)
   ->  Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..8.97 rows=39 width=8)
         Index Cond: (id < 40)
   ->  Materialize  (cost=0.00..85.55 rows=9 width=8)
         ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=9 width=8)
               Filter: (id < 10)
(7 rows)
```

(3) Indexed nested loop join with outer index scan

```
testdb=# SET enable_hashjoin TO off;
SET
testdb=# SET enable_mergejoin TO off;
SET
testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_d AS d WHERE a.id = d.id AND a.id <  40;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.57..173.06 rows=20 width=16)
   ->  Index Scan using tbl_a_pkey on tbl_a a  (cost=0.29..8.97 rows=39 width=8)
         Index Cond: (id < 40)
   ->  Index Scan using tbl_d_pkey on tbl_d d  (cost=0.28..4.20 rows=1 width=8)
         Index Cond: (id = a.id)
(5 rows)
```

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 $O(N_{outer}\log_2(N_{outer}) + N_{inner}\log_2(N_{inner}))$, where N_{outer} and N_{inner} are the number of tuples of the outer and inner tables, respectively. The run cost is $O(N_{outer} + N_{inner})$.

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

3.5.2.1. Merge Join

Figure 3.23 shows a conceptual illustration of a merge join.

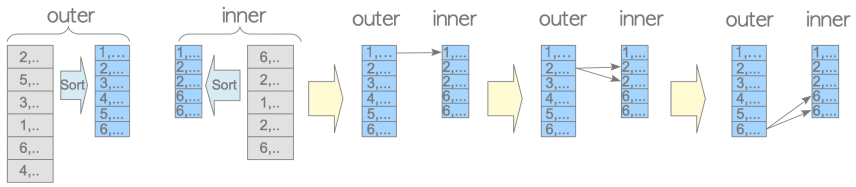

Figure 3.23. Merge join.

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.

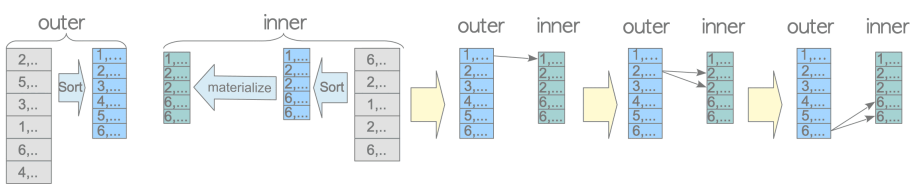

Figure 3.24. Materialized merge join.

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  Merge Join  (cost=10466.08..10578.58 rows=5000 width=2064)
 5    Merge Cond: (a.id = b.id)
 6    ->  Sort  (cost=6708.39..6733.39 rows=10000 width=1032)
 7          Sort Key: a.id
 8          ->  Seq Scan on tbl_a a  (cost=0.00..1529.00 rows=10000 width=1032)
 9    ->  Materialize  (cost=3757.69..3782.69 rows=5000 width=1032)
10          ->  Sort  (cost=3757.69..3770.19 rows=5000 width=1032)
11                Sort Key: b.id
12                ->  Seq Scan on tbl_b b  (cost=0.00..1193.00 rows=5000 width=1032)
13 (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.

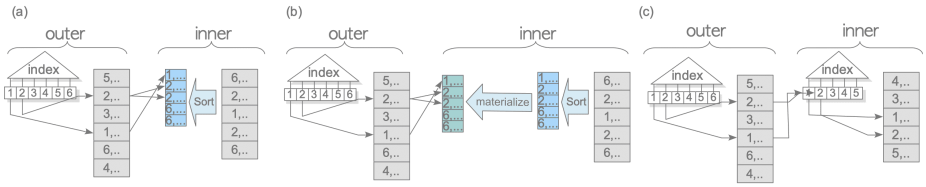

Figure 3.25. The three variations of the merge join with an outer index scan.

The results of these joins' EXPLAIN are shown below:

(a) Merge join with outer index scan

```
testdb=# SET enable_hashjoin TO off;
SET
testdb=# SET enable_nestloop TO off;
SET
testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_b AS b WHERE c.id = b.id AND b.id < 1000;
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Merge Join  (cost=135.61..322.11 rows=1000 width=16)
   Merge Cond: (c.id = b.id)
   ->  Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..318.29 rows=10000 width=8)
   ->  Sort  (cost=135.33..137.83 rows=1000 width=8)
         Sort Key: b.id
         ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=1000 width=8)
               Filter: (id < 1000)
(7 rows)
```

(b) Materialized merge join with outer index scan

```
testdb=# SET enable_hashjoin TO off;
SET
testdb=# SET enable_nestloop TO off;
SET
testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_b AS b WHERE c.id = b.id AND b.id < 4500;
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Merge Join  (cost=421.84..672.09 rows=4500 width=16)
   Merge Cond: (c.id = b.id)
   ->  Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..318.29 rows=10000 width=8)
   ->  Materialize  (cost=421.55..444.05 rows=4500 width=8)
         ->  Sort  (cost=421.55..432.80 rows=4500 width=8)
               Sort Key: b.id
               ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=4500 width=8)
                     Filter: (id < 4500)
(8 rows)
```

(c) Indexed merge join with outer index scan

```
testdb=# SET enable_hashjoin TO off;
SET
testdb=# SET enable_nestloop TO off;
SET
testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_d AS d WHERE c.id = d.id AND d.id < 1000;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Merge Join  (cost=0.57..226.07 rows=1000 width=16)
   Merge Cond: (c.id = d.id)
   ->  Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..318.29 rows=10000 width=8)
   ->  Index Scan using tbl_d_pkey on tbl_d d  (cost=0.28..41.78 rows=1000 width=8)
         Index Cond: (id < 1000)
(5 rows)
```

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 Figures 3.26 and 3.27.

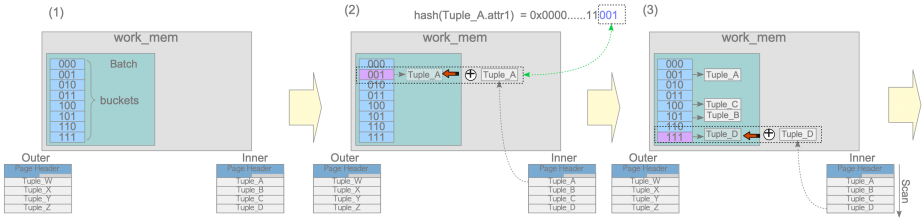

Figure 3.26. The build phase in the in-memory hash join.

- (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 notatio, 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: ⊕

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

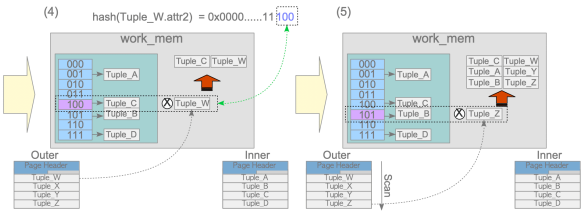

Figure 3.27. The probe phase in the in-memory hash join.

- (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: ⊗

- (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 3.28 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.

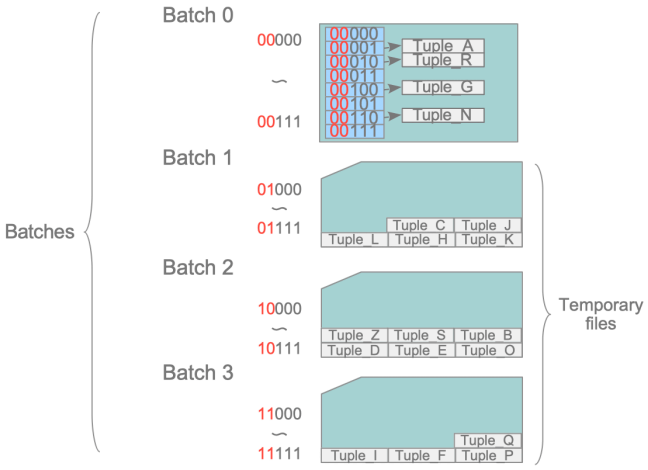

Figure 3.28. Multiple batches in hybrid hash join.

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 Figures 3.29 to 3.32.

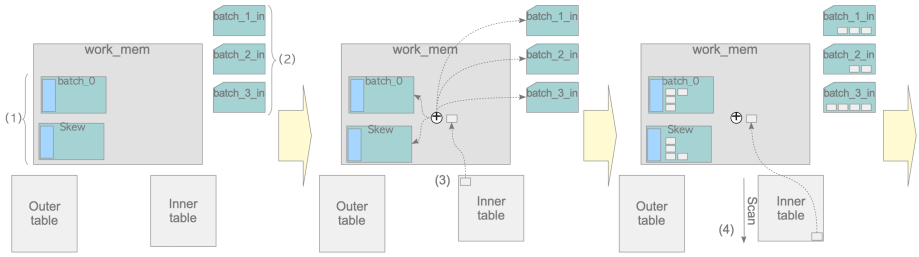

Figure 3.29. The build phase of the hybrid hash join in the first round.

- (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.

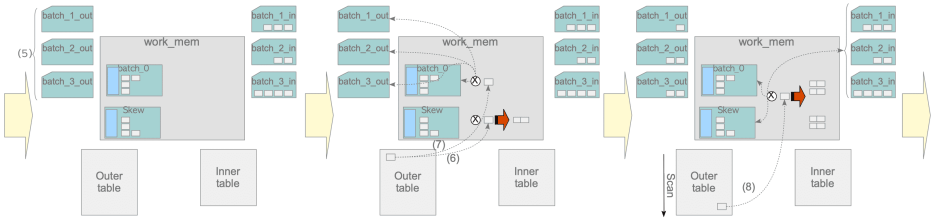

Figure 3.30. The probe phase of the hybrid hash join in the first round.

- (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.

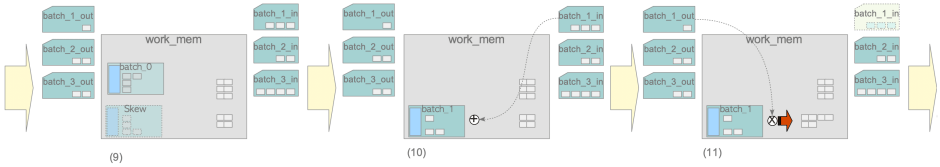

Figure 3.31. The build and probe phases in the second round.

- (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'.

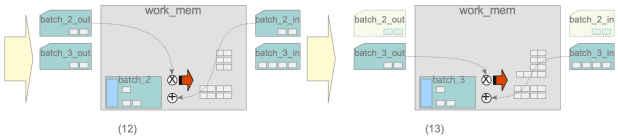

Figure 3.32. The build and probe phases in the third and the last rounds.

- (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.aid BETWEEN 100 AND
3  1000;
4                                                QUERY PLAN
5  ------------------------------------------------------------------------------------------------
6  ------
7   Hash Join  (cost=1.88..51.93 rows=865 width=461)
8     Hash Cond: (a.bid = b.bid)
9     ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts a  (cost=0.43..47.73 rows=865
10 width=97)
11           Index Cond: ((aid >= 100) AND (aid <= 1000))
       ->  Hash  (cost=1.20..1.20 rows=20 width=364)
             ->  Seq Scan on pgbench_branches b  (cost=0.00..1.20 rows=20 width=364)
   (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.

JoinPath

```
typedef JoinPath NestPath;

/*
 * JoinType -
 *    enums for types of relation joins
 *
 * JoinType determines the exact semantics of joining two relations using
 * a matching qualification.  For example, it tells what to do with a tuple
 * that has no match in the other relation.
 *
 * This is needed in both parsenodes.h and plannodes.h, so put it here...
 */
typedef enum JoinType
{
    /*
     * The canonical kinds of joins according to the SQL JOIN syntax. Only
     * these codes can appear in parser output (e.g., JoinExpr nodes).
     */
    JOIN_INNER,                 /* matching tuple pairs only */
    JOIN_LEFT,                  /* pairs + unmatched LHS tuples */
    JOIN_FULL,                  /* pairs + unmatched LHS + unmatched RHS */
    JOIN_RIGHT,                 /* pairs + unmatched RHS tuples */

    /*
     * Semijoins and anti-semijoins (as defined in relational theory) do not
     * appear in the SQL JOIN syntax, but there are standard idioms for
     * representing them (e.g., using EXISTS).  The planner recognizes these
     * cases and converts them to joins.  So the planner and executor must
     * support these codes.  NOTE: in JOIN_SEMI output, it is unspecified
     * which matching RHS row is joined to.  In JOIN_ANTI output, the row is
     * guaranteed to be null-extended.
     */
    JOIN_SEMI,                  /* 1 copy of each LHS row that has match(es) */
    JOIN_ANTI,                  /* 1 copy of each LHS row that has no match */
    JOIN_RIGHT_ANTI,            /* 1 copy of each RHS row that has no match */

    /*
     * These codes are used internally in the planner, but are not supported
     * by the executor (nor, indeed, by most of the planner).
     */
    JOIN_UNIQUE_OUTER,          /* LHS path must be made unique */
    JOIN_UNIQUE_INNER           /* RHS path must be made unique */

    /*
     * 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;
```

MergePath

```
/*
```

```
 * 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 palloc 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;
```

HashPath

```
/*
 * 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;
```

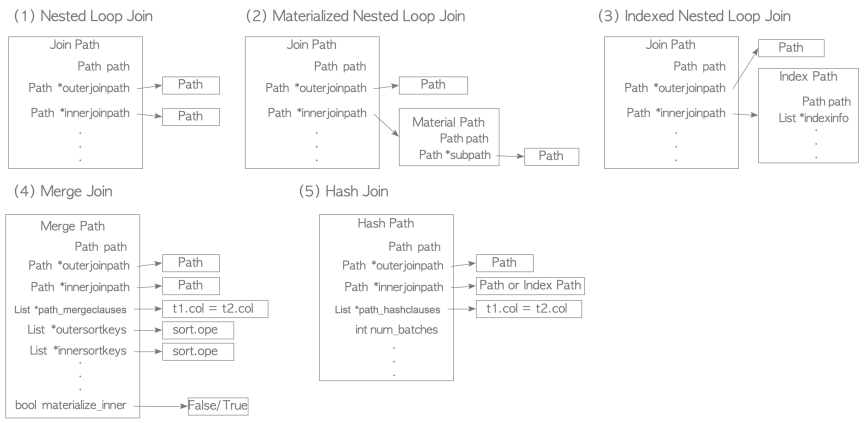

Figure 3.33. Join access paths.

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`.

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;
```

NestedLoopNode

```
/* ----------------
 *      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 Param to set */
    Var        *paramval;       /* outer-relation Var to assign to Param */
} NestLoopParam;
```

MergeJoinNode

```
/* ----------------
 *      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;
```

HashJoinNode

```
/* ----------------
 *      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 ℹ below.

ℹ 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.

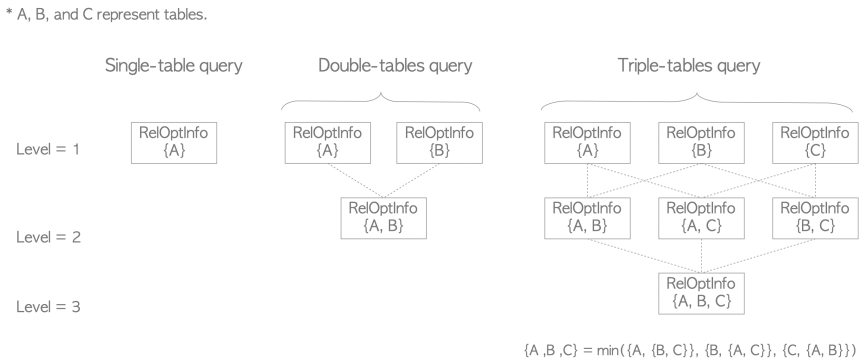

Figure 3.34. How to get the cheapest access path using dynamic programming.

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.

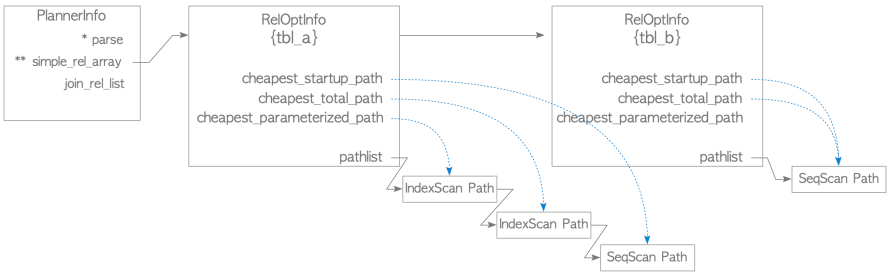

Figure 3.35. The PlannerInfo and RelOptInfo after processing in Level 1.

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 Figure 3.36.

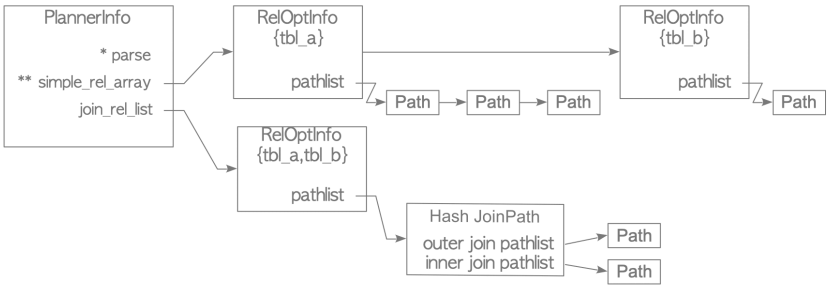

Figure 3.36. The PlannerInfo and RelOptInfo after processing in Level 2.

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 Figure 3.36.

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.

```
testdb=# EXPLAIN  SELECT * FROM tbl_b AS b, tbl_c AS c WHERE c.id = b.id AND b.data < 400;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=90.50..277.00 rows=400 width=16)
   Hash Cond: (c.id = b.id)
   ->  Seq Scan on tbl_c c  (cost=0.00..145.00 rows=10000 width=8)
   ->  Hash  (cost=85.50..85.50 rows=400 width=8)
         ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=400 width=8)
               Filter: (data < 400)
(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

$$\{\text{tbl\_a}, \text{tbl\_b}, \text{tbl\_c}\} = \min(\{\text{tbl\_a}, \{\text{tbl\_b}, \text{tbl\_c}\}\}, \{\text{tbl\_b}, \{\text{tbl\_a}, \text{tbl\_c}\}\}, \{\text{tbl\_c}, \{\text{tbl\_a}, \text{tbl\_b}\}\}).$$

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

In the RelOptInfo onject {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)
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)
```

Outer relation of Indexed Nested Loop Join

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.

3.7. PARALLEL QUERY

Parallel Query, supported from version 9.6, is a feature that processes a single query using multiple processes (Workers).

For instance, if certain conditions are met, the PostgreSQL process that executes the query becomes the Leader and starts up to Worker processes (maximum number is max_parallel_workers_per_gather). Each Worker process then performs scan processing, and returns the results sequentially to the Leader process. The Leader process aggregates the results returned from the Worker processes.

Figure 3.38 illustrates how a sequential scan query is processed by two Worker processes.

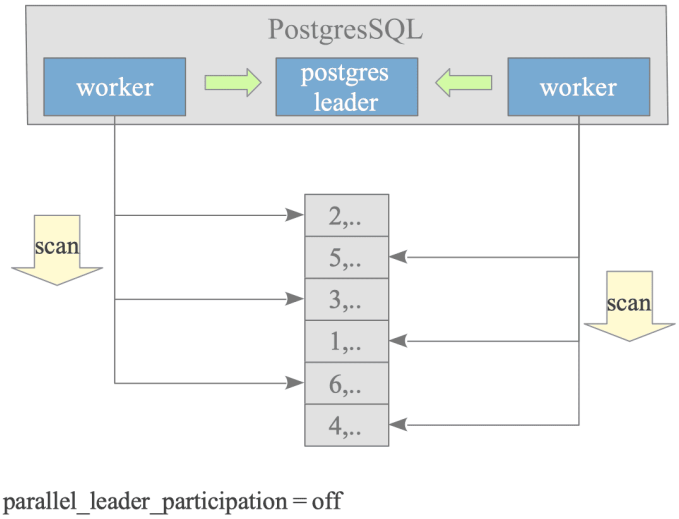

Figure 3.38. Parallel Query Concept.

The configuration parameter parallel_leader_participation, introduced in version 14, allows the Leader process to also process queries while waiting for responses from Workers. The default is `on`.

However, to simplify the explanation and diagrams, we will explain this under the assumption that the Leader process does not process queries (i.e., parallel_leader_participation is `off`).

Parallel Query is gradually improved. Table 3.2 highlights key points from the official documentation's release notes about Parallel Query.

Table 3.2 Release notes related to Parallel Query. (cited from the official document)

Version	Description
9.6	• Seq Scan. • Nested Loop Join and Hash Join.
10	• Merge Join. • B-tree Index Scan, Bitmap Heap Scan. • Allow non-correlated subqueries.
11	• CREATE INDEX can now use parallel processing while building a B-tree index. • Allow hash joins to be performed in parallel using a shared hash table. • Allow UNION to run each SELECT in parallel if the individual SELECTs cannot be parallelized. • Allow parallelization of commands CREATE TABLE ... AS, SELECT INTO, and CREATE MATERIALIZED VIEW.
12	• Allow parallelized queries when in SERIALIZABLE isolation mode.
14	• Allow a query referencing multiple foreign tables to perform foreign table scans in parallel. • Allow REFRESH MATERIALIZED VIEW to use parallelism.
15	• Allow SELECT DISTINCT to be parallelized. • Allow a query referencing multiple foreign tables to perform parallel foreign table scans in more cases. • Allow parallel commit on postgres_fdw servers.
16	• Allow parallelization of FULL and internal right OUTER hash joins. • Allow postgres_fdw to do aborts in parallel.

Note that Parallel Query is primarily READ-ONLY and does not yet support cursor operations.

In the following, we will first provide an overview of Parallel Query, then explore Join operations, and finally explain how aggregate functions are calculated in parallel queries.

3.7.1. Overview

Figure 3.39 briefly describes how parallel query performs in PostgreSQL.

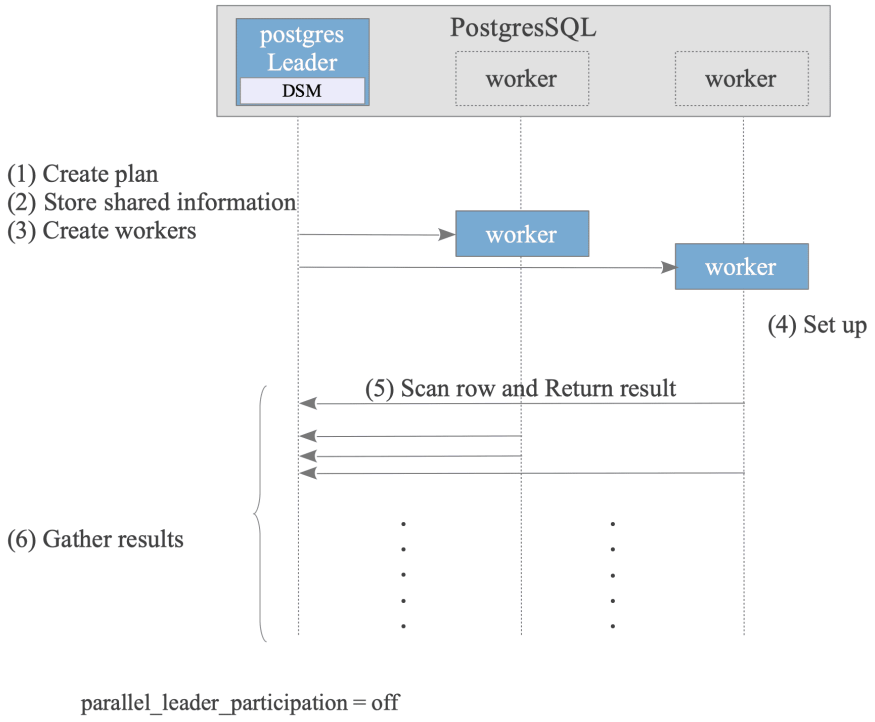

Figure 3.39. How Parallel Query Performs.

1. Leader Creates Plan:
   The optimizer creates a plan that can be executed in parallel.
2. Leader Stores Shared information:
   To execute the plan on both the Leader and Worker nodes, the Leader stores necessary information in its Dynamic Shared Memory (DSM) area. For more details, see the following explanation.
3. Leader Creates Workers.
4. Worker Sets up State:
   Each worker reads the stored shared information to sets up its state, ensuring a consistent execution environment with the Leader.
5. Worker Scans rows and Returns results.
6. Leader Gathers results.
7. After the query finishes, the Workers are terminated, and the Leader releases the DSM area.

In Parallel Query processing, the Leader and Worker processes communicate Dynamic Shared Memory (DSM) area.

The Leader process allocates memory space on demand (i.e., it can be allocated as needed and released when no longer required). Worker processes can then read and write data to these memory regions as if they were their own.

In this section, to demonstrate using concrete examples, we use Table `d` as follows:

```
testdb=# CREATE TABLE d (id double precision, data int);
CREATE TABLE
testdb=# INSERT INTO d SELECT i::double precision, (random()*1000)::int FROM generate_series(1, 1000000)
AS i;
INSERT 0 1000000
testdb=# ANALYZE;
ANALYZE
```

3.7.1.1. Creating Parallel Plan

Parallel Query is not always considered. If the table size to be scanned is greater than or equal to min_parallel_table_scan_size (default is 8MB), or if the index size to be scanned is greater than or equal to min_parallel_index_scan_size (default is 512kB), the optimizer will consider Parallel Query.

Here is the simplest plan in parallel query:

```
testdb=# EXPLAIN SELECT * FROM d WHERE id BETWEEN 1 AND 100;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..16609.10 rows=1 width=12)
   Workers Planned: 2
   ->  Parallel Seq Scan on d  (cost=0.00..15609.00 rows=1 width=12)
         Filter: ((id >= '1'::double precision) AND (id <= '100'::double precision))
(4 rows)
```

As shown above, the simplest plan consists of a `Gather` node and a `Seq Scan` node.

Figure 3.40 shows the plan tree of the above query:

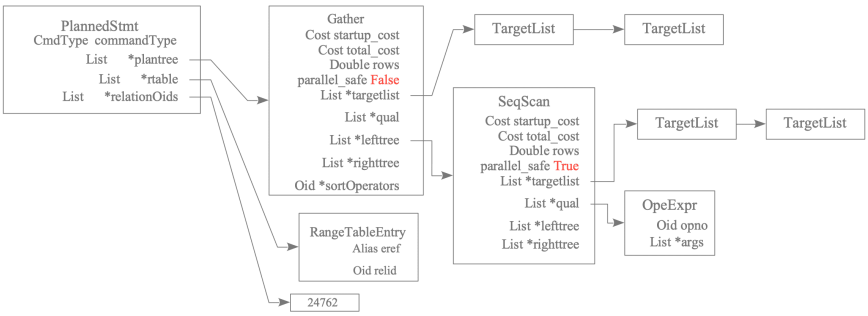

Figure 3.40. Leader's Plan Tree

The Gather node is specific to Parallel Query and collects the results from the Workers. In addition to Gather, Parallel Query has the following specific nodes:

- GatherMerge
- Append and AppendMerge (See Official Document.)
- Finalize/Partial Aggregate (See Section 3.7.3.)

The nodes executed by Workers are generally the subplan tree below the Gather node, and the `parallel_safe` value is set to `True`. In the above example, the Seq Scan node and below, which is under the Gather node, are executed by Workers.

3.7.1.2. Storing Shared Information

To execute the query collaboratively with the Workers, the Leader stores information to be shared with the Workers in its DSM (Dynamic Shared Memory) area.

The Leader and Workers share two types of information: execution state and query.

- Execution State:
  The environment information necessary for the Leader and Workers to execute the same query consistently. See README.parallel in details. This includes information such as:
  - All configuration parameters (GUC)
  - The transaction snapshot, the current subtransaction's XID. (For more details on these, see Chapter 5.)
  - The set of libraries dynamically loaded by dfmgr.c.

  This information is stored by InitializeParallelDSM().
- Query:
  This includes information such as:
  - PlannedStmt
  - ParamListInfo
  - The Description structures of nodes executed by Worker. For example, the SeqScan node uses `ParallelTableScanDesc`; the IndexScan node uses `ParallelIndexScanDesc`. See ExecParallelInitializeDSM() for detail.
  - The instrumentation and usage information for reporting purposes.

  This information is stored by ExecInitParallelPlan().

Additionally, the Leader creates a TupleQueue (based on tqueue.c) on DSM to read the Worker's results.

3.7.1.3. Creating Workers

The number of Workers planned in the query plan may differ from the actual number of Workers running. This is because the maximum number of Workers is limited by max_parallel_workers, and there may not be enough available Workers due to other parallel query processes.

The EXPLAIN ANALYZE command shows both the planned and launched number of Workers.

```
testdb=# EXPLAIN ANALYZE SELECT * FROM d WHERE id BETWEEN 1 AND 100;
                                  QUERY PLAN
```

```
-------------------------------------------------------------------------------------------------------------
----------
 Gather  (cost=1000.00..16609.10 rows=1 width=12) (actual time=60.035..60.994 rows=100 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on d  (cost=0.00..15609.00 rows=1 width=12) (actual time=31.073..52.248 rows=50
loops=2)
         Filter: ((id >= '1'::double precision) AND (id <= '100'::double precision))
         Rows Removed by Filter: 499950
 Planning Time: 0.265 ms
 Execution Time: 61.012 ms
(8 rows)
```

3.7.1.4. Setting Up Workers

Upon startup, a Worker reads the shared Execution State and Query information prepared by the Leader.

By setting the Execution State information, the Worker can execute the query in the same environment as the Leader.

When reading the Query information, the ExecSerializePlan() function is used to generate and save a plan tree for the Worker based on the PlannedStmt. Specifically, a new sub-plan tree (typically a Gather node sub-plan tree) where parallel_safe is True is generated as the Worker's plan tree.

Figure 3.41 shows the plan tree of the Worker.

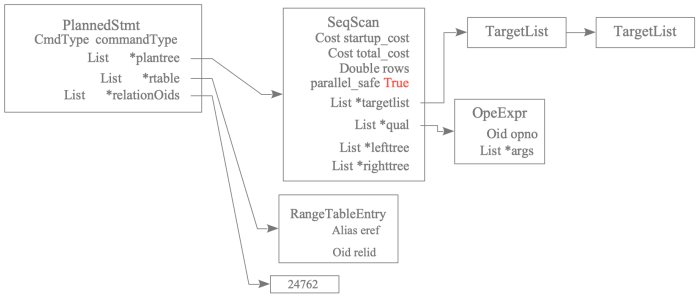

Figure 3.41. Worker's Plan Tree Generated from Leader's Plan Tree

3.7.1.5. Scanning Rows and Returning Results

As discussed in ℹ"Scanning Functions" in Section 3.4.1, the Executor's methods for accessing individual data tuples are highly abstracted. This principle extends to Parallel Queries.

Although a detailed explanation is omitted, since the Leader and Workers share the query execution environment via DSM, a single sequential scan can be executed in parallel by the Leader and Workers calling the SeqNext() function.

Similarly, results are returned to the Gather node via a TupleQueue on DSM.

3.7.1.6. Gathering Results

The Gather node, a Parallel Query specific node, gathers the results returned by the Workers.

3.7.2. Parallel Join

Parallel Query in PostgreSQL supports nested-loop joins, merge joins, and hash joins.

In this section, we will use the following tables: `d` and `f`:

```
testdb=# CREATE TABLE d (id double precision, data int);
CREATE TABLE
testdb=# INSERT INTO d SELECT i::double precision, (random()*1000)::int FROM generate_series(1, 1000000)
AS i;
INSERT 0 1000000
testdb=# CREATE INDEX d_id_idx ON d (id);
CREATE INDEX
testdb=# CREATE TABLE f (id double precision, data int);
CREATE TABLE
testdb=# INSERT INTO f SELECT i::double precision, (random()*1000)::int FROM generate_series(1,
10000000) AS i;
INSERT 0 10000000
testdb=# \d d
                    Table "public.d"
 Column |       Type       | Collation | Nullable | Default
--------+------------------+-----------+----------+---------
 id     | double precision |           |          |
 data   | integer          |           |          |
Indexes:
    "d_id_idx" btree (id)

testdb=# \d f
                    Table "public.f"
 Column |       Type       | Collation | Nullable | Default
--------+------------------+-----------+----------+---------
 id     | double precision |           |          |
 data   | integer          |           |          |
```

```
testdb=# ANALYZE;
ANALYZE
```

3.7.2.1. Nested Loop Join

In a nested loop join, the inner table is always processed for all rows, not in parallel.

Therefore, for example, in a materialized nested loop join, each Worker must materialize the inner table independently, which is inefficient.

```
testdb=# SET enable_nestloop = ON;
SET
testdb=# SET enable_mergejoin = OFF;
SET
testdb=# SET enable_hashjoin = OFF;
SET

testdb=# EXPLAIN SELECT * FROM d, f WHERE d.data = f.data AND f.id < 10000;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..97163469.29 rows=9651513 width=24)
   Workers Planned: 2
   ->  Nested Loop  (cost=0.00..96197317.99 rows=4825756 width=24)
         Join Filter: (d.data = f.data)
         ->  Parallel Seq Scan on f  (cost=0.00..121935.99 rows=4831 width=12)
               Filter: (id < '10000'::double precision)
         ->  Materialize  (cost=0.00..27992.00 rows=1000000 width=12)
               ->  Seq Scan on d  (cost=0.00..18109.00 rows=1000000 width=12)
(8 rows)
```

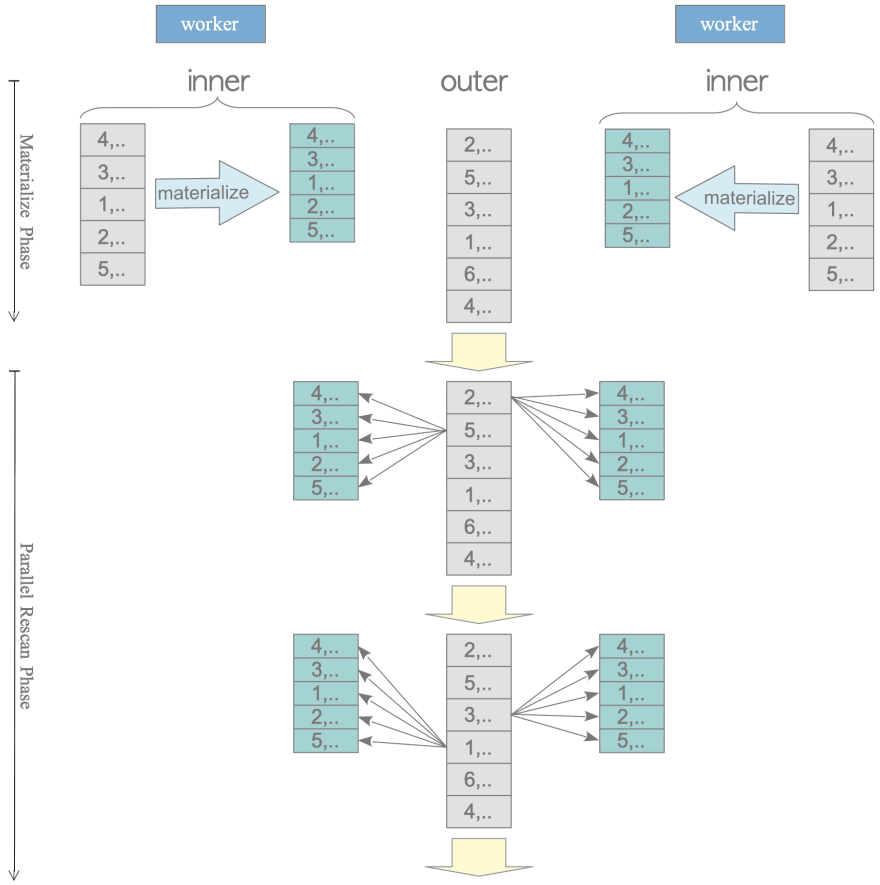

Figure 3.42. Materialize Nested Loop Join in Parallel Query.

On the other hand, in an Indexed Nested Loop Join, the inner table can be scanned in parallel by the Workers, making it more efficient.

```
testdb=# EXPLAIN SELECT * FROM d, f WHERE d.id = f.id AND f.id < 10000;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.42..142818.71 rows=967 width=24)
   Workers Planned: 2
   ->  Nested Loop  (cost=0.42..141722.01 rows=484 width=24)
         ->  Parallel Seq Scan on f  (cost=0.00..121935.99 rows=4831 width=12)
               Filter: (id < '10000'::double precision)
         ->  Index Scan using d_id_idx on d  (cost=0.42..4.09 rows=1 width=12)
               Index Cond: (id = f.id)
(7 rows)
```

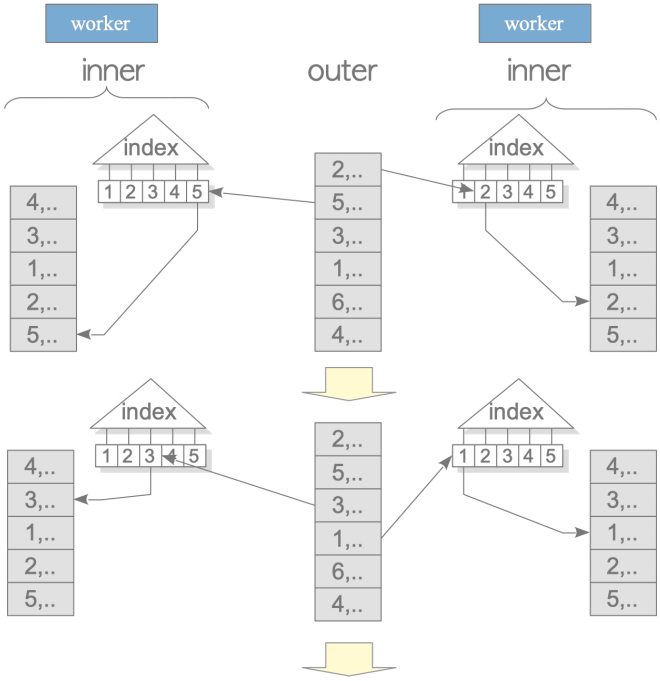

Figure 3.43. Indexed Nested Loop Join in Parallel Query.

3.7.2.2. Merge Join

Like nested loop joins, a merge join always processes the inner table for all rows, not in parallel. Therefore, each Worker performs the sorting process of the inner table independently.

However, if the inner table is accessed using an index scan, the join operation can be performed efficiently, similar to an Indexed Nested Loop Join.

```
testdb=# SET enable_nestloop = OFF;
SET
testdb=# SET enable_mergejoin = ON;
SET
testdb=# SET enable_hashjoin = OFF;
SET
testdb=# EXPLAIN SELECT * FROM d, f WHERE d.id = f.id AND d.id < 100000;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Gather  (cost=837387.83..853944.33 rows=97361 width=24)
   Workers Planned: 2
   ->  Merge Join  (cost=836387.83..843208.23 rows=48680 width=24)
         Merge Cond: (f.id = d.id)
         ->  Sort  (cost=836385.61..848880.80 rows=4998079 width=12)
               Sort Key: f.id
               ->  Parallel Seq Scan on f  (cost=0.00..109440.79 rows=4998079 width=12)
         ->  Index Scan using d_id_idx on d  (cost=0.42..3569.24 rows=97361 width=12)
               Index Cond: (id < '100000'::double precision)
(9 rows)
```

3.7.2.3. Hash Join

In versions 10 and 9.6, during a Parallel Query Hash Join, each Worker performs the hash table build process for the inner table independently.

```
testdb=# SET enable_nestloop = OFF;
SET
testdb=# SET enable_mergejoin = OFF;
SET
testdb=# SET enable_hashjoin = ON;
SET
testdb=# SET enable_parallel_hash = OFF;
SET
testdb=# EXPLAIN SELECT * FROM d, f WHERE d.id = f.id;
                                    QUERY PLAN
-----------------------------------------------------------------------------
 Gather  (cost=36492.00..323368.59 rows=1000000 width=24)
   Workers Planned: 2
   ->  Hash Join  (cost=35492.00..222368.59 rows=500000 width=24)
         Hash Cond: (f.id = d.id)
         ->  Parallel Seq Scan on f  (cost=0.00..109440.79 rows=4998079 width=12)
         ->  Hash  (cost=18109.00..18109.00 rows=1000000 width=12)
               ->  Seq Scan on d  (cost=0.00..18109.00 rows=1000000 width=12)
(7 rows)
```

Parallel hash join was supported in version 11 (enable_parallel_hash, enabled by default). With this feature, hash tables are created for each Worker during the build phase and are shared between workers, making the build phase more

efficient.

```
testdb=# SET enable_parallel_hash = ON;
SET
testdb=# EXPLAIN SELECT * FROM d, f WHERE d.id = f.id;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Gather  (cost=22801.00..304736.59 rows=1000000 width=24)
   Workers Planned: 2
   ->  Parallel Hash Join  (cost=21801.00..203736.59 rows=500000 width=24)
         Hash Cond: (f.id = d.id)
         ->  Parallel Seq Scan on f  (cost=0.00..109440.79 rows=4998079 width=12)
         ->  Parallel Hash  (cost=13109.00..13109.00 rows=500000 width=12)
               ->  Parallel Seq Scan on d  (cost=0.00..13109.00 rows=500000 width=12)
(7 rows)
```

3.7.3. Parallel Aggregate

Almost aggregate functions in PostgreSQL can be also processed in parallel. Whether an aggregate function can be parallelized depends on whether `Partial Mode` is set to `YES` in Official Document.

When the predicted number of target rows is small:

1. Scanned rows from each Worker, processed by the parallel (seq) scan node, are gathered at the gather node.
2. These gathered rows are then aggregated at the aggregate node.

See the following examples:

```
testdb=# EXPLAIN SELECT avg(id) FROM d where id BETWEEN 1 AND 10;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate  (cost=16609.10..16609.11 rows=1 width=8)
   ->  Gather  (cost=1000.00..16609.10 rows=1 width=8)
         Workers Planned: 2
         ->  Parallel Seq Scan on d  (cost=0.00..15609.00 rows=1 width=8)
               Filter: ((id >= '1'::double precision) AND (id <= '10'::double precision))
(5 rows)

testdb=# EXPLAIN SELECT var_pop(id) FROM d where id BETWEEN 1 AND 10;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate  (cost=16609.10..16609.11 rows=1 width=8)
   ->  Gather  (cost=1000.00..16609.10 rows=1 width=8)
         Workers Planned: 2
         ->  Parallel Seq Scan on d  (cost=0.00..15609.00 rows=1 width=8)
               Filter: ((id >= '1'::double precision) AND (id <= '10'::double precision))
(5 rows)
```

When the predicted number of target rows is large:

1. Scanned rows from each Worker, processed by the parallel sequential scan node, are partially aggregated at the aggregate node.
2. These intermediate aggregated results are gathered at the gather node.
3. They are then finally aggregated at the finalize aggregate node.

See the following examples:

```
testdb=# EXPLAIN SELECT avg(id) FROM d where data > 100;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=16485.14..16485.15 rows=1 width=8)
   ->  Gather  (cost=16484.93..16485.14 rows=2 width=32)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=15484.93..15484.94 rows=1 width=32)
               ->  Parallel Seq Scan on d  (cost=0.00..14359.00 rows=450371 width=8)
                     Filter: (data > 100)
(6 rows)

testdb=# EXPLAIN  SELECT var_pop(id) FROM d where data > 100;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=16485.14..16485.15 rows=1 width=8)
   ->  Gather  (cost=16484.93..16485.14 rows=2 width=32)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=15484.93..15484.94 rows=1 width=32)
               ->  Parallel Seq Scan on d  (cost=0.00..14359.00 rows=450371 width=8)
                     Filter: (data > 100)
(6 rows)
```

Multiple sums, averages, and variances can be calculated using the following formulas:

$$
\begin{aligned}
S_n &= S_{n_1} + S_{n_2} \\
A_n &= \frac{1}{n_1 + n_2}(S_{n_1} + S_{n_2}) \\
V_n &= (V_{n_1} + V_{n_2}) + \frac{n_1 n_2}{n_1 + n_2}\left(\frac{S_{n_1}}{n_1} - \frac{S_{n_2}}{n_2}\right)^2
\end{aligned}
$$

The finalize aggregate node aggregates the intermediate results using the formulas above.

For three or more values, the process can be repeated iteratively, adding one value at a time.

Combining Multiple Variances

Given:

- A dataset: $\{x_i | 1 \leq i \leq n = n_1 + n_2\}$
- The overall average: $A_n = \sum_{i=1}^{n_1+n_2} x_i$
- The sum of the first part of the dataset: $S_{n_1} = \sum_{i=1}^{n_1} x_i$
- The variance of the first part: $V_{n_1} = \sum_{i=1}^{n_1} (x_i - \frac{S_{n_1}}{n_1})^2$
- The sum of the second part of the dataset: $S_{n_2} = \sum_{i=1+n_1}^{n_1+n_2} x_i$
- The variance of the second part: $V_{n_2} = \sum_{i=1+n_1}^{n_1+n_2} (x_i - \frac{S_{n_2}}{n_2})^2$

To Prove:

The variance of the entire dataset, V_n, can be expressed in terms of V_{n_1}, V_{n_2}, S_{n_1}, and S_{n_2} as follows:

$$V_n = V_{n_1} + V_{n_2} + \frac{n_1 n_2}{n_1 + n_2} \left(\frac{S_{n_1}}{n_1} - \frac{S_{n_2}}{n_2} \right)^2$$

Proof:

$$\begin{aligned}
V_n &= \sum_{i=1}^{n} (x_i - A_n)^2 = \sum_{i=1}^{n_1} (x_i - A_n)^2 + \sum_{i=1+n_1}^{n_1+n_2} (x_i - A_n)^2 \\
&= \sum_{i=1}^{n_1} \left(x_i - (\frac{S_{n_1} + S_{n_2}}{n_1 + n_2}) \right)^2 + \sum_{i=1+n_1}^{n_1+n_2} \left(x_i - (\frac{S_{n_1} + S_{n_2}}{n_1 + n_2}) \right)^2 \\
&= \sum_{i=1}^{n_1} \left((x_i - \frac{S_{n_1}}{n_1}) + \frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right)^2 + \sum_{i=1+n_1}^{n_1+n_2} \left((x_i - \frac{S_{n_2}}{n_2}) + \frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right)^2 \\
&= \sum_{i=1}^{n_1} \left(x_i - \frac{S_{n_1}}{n_1} \right)^2 + 2 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right) \sum_{i=1}^{n_1} \left(x_i - \frac{S_{n_1}}{n_1} \right) + \sum_{i=1}^{n_1} \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right)^2 \\
&\quad + \sum_{i=1+n_1}^{n_1+n_2} \left(x_i - \frac{S_{n_2}}{n_2} \right)^2 + 2 \left(\frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right) \sum_{i=1+n_1}^{n_1+n_2} \left(x_i - \frac{S_{n_2}}{n_2} \right) + \sum_{i=1+n_1}^{n_1+n_2} \left(\frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right)^2 \\
&= V_{n_1} + 2 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right) \left(\sum_{i=1}^{n_1} x_i - n_1 \frac{S_{n_1}}{n_1} \right) + n_1 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right)^2 \\
&\quad + V_{n_2} + 2 \left(\frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right) \left(\sum_{i=1+n_1}^{n_1+n_2} x_i - n_2 \frac{S_{n_2}}{n_2} \right) + n_2 \left(\frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right)^2 \\
&= V_{n_1} + 2 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right) (S_{n_1} - S_{n_1}) + n_1 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right)^2 \\
&\quad + V_{n_2} + 2 \left(\frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right) (S_{n_2} - S_{n_2}) + n_2 \left(\frac{(-1)(S_{n_1} n_2 - S_{n_2} n_1)}{(n_1 + n_2) n_2} \right)^2 \\
&= V_{n_1} + 2 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right) \cdot 0 + n_1 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_1} \right)^2 \\
&\quad + V_{n_2} + 2 \left(\frac{S_{n_2} n_1 - S_{n_1} n_2}{(n_1 + n_2) n_2} \right) \cdot 0 + n_2 \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{(n_1 + n_2) n_2} \right)^2 \\
&= V_{n_1} + V_{n_2} + \frac{1}{n_1} \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{n_1 + n_2} \right)^2 + \frac{1}{n_2} \left(\frac{S_{n_1} n_2 - S_{n_2} n_1}{n_1 + n_2} \right)^2 \\
&= V_{n_1} + V_{n_2} + \left(\frac{1}{n_1} + \frac{1}{n_2} \right) \left(\frac{n_1 n_2}{n_1 + n_2} \right)^2 \left(\frac{S_{n_1}}{n_1} - \frac{S_{n_2}}{n_2} \right)^2 \\
&= V_{n_1} + V_{n_2} + \frac{n_1 n_2}{n_1 + n_2} \left(\frac{S_{n_1}}{n_1} - \frac{S_{n_2}}{n_2} \right)^2
\end{aligned}$$

4. 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.

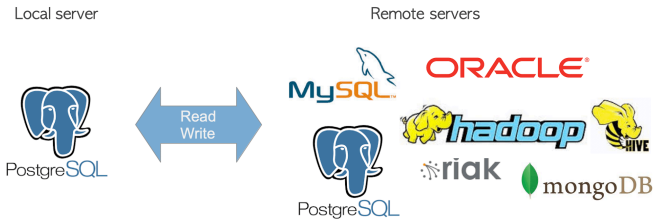

Figure 4.1. Basic concept of FDW.

After installing the necessary extension and configuring the appropriate settings, foreign tables on remote servers can be accessed.

For example, consider two remote servers, one running PostgreSQL (with a table named foreign_pg_tbl) and another running MySQL (with a table named foreign_my_tbl). These foreign tables can be accessed from the local server using queries such as the following:

```
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
```

Similar to local tables, FDWs also enable join operations between foreign tables stored on different servers.

```
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.

Chapter Contents

4.1. OVERVIEW

To utilize the FDW feature, the appropriate extension must be installed, and 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.

Figure 4.2 briefly describes how FDW performs in PostgreSQL.

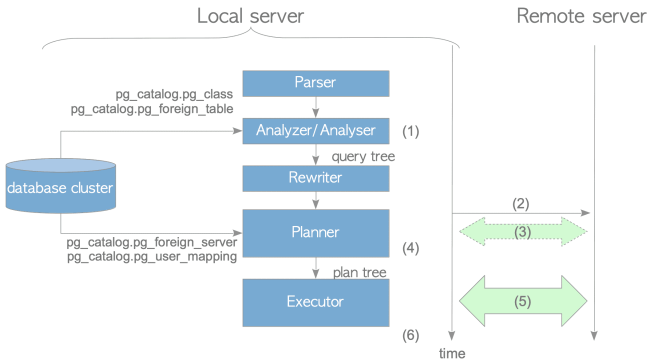

Figure 4.2. How FDWs perform.

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 **deparesing**.
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. 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.

4.1.2. 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.

4.1.3. 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.

4.1.4. Deparesing

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, Figure 4.3 shows the plan tree of the following SELECT statement:

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

Figure 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.

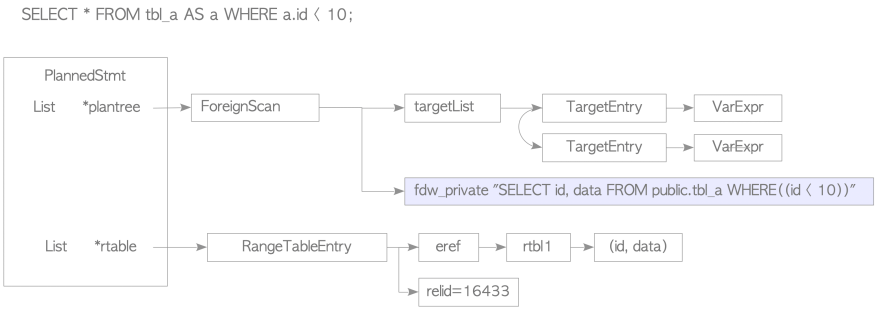

Figure 4.3. Example of the plan tree that scans a foreign table.

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.

4.1.5. 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 (Figure 4.4).

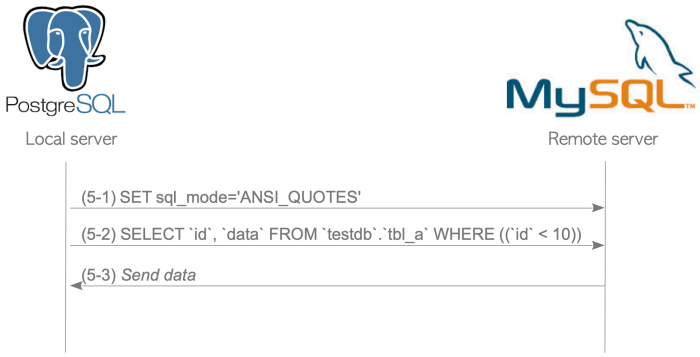

Figure 4.4. Typical sequence of SQL statements to execute a SELECT query in mysql_fdw.

- (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.

Actual log of the remote server

```
mysql> SELECT command_type,argument FROM mysql.general_log;
+--------------+-------------------------------------------------------------------------+
| command_type | argument                                                                |
+--------------+-------------------------------------------------------------------------+
```

```
... snip ...

| Query       | SET sql_mode='ANSI_QUOTES'                                                  |
| Prepare     | SELECT `id`, `data` FROM `localdb`.`tbl_a` WHERE ((`id` < 10))              |
| Close stmt  |                                                                             |
+-------------+-----------------------------------------------------------------------------+
```

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 (Figure 4.5).

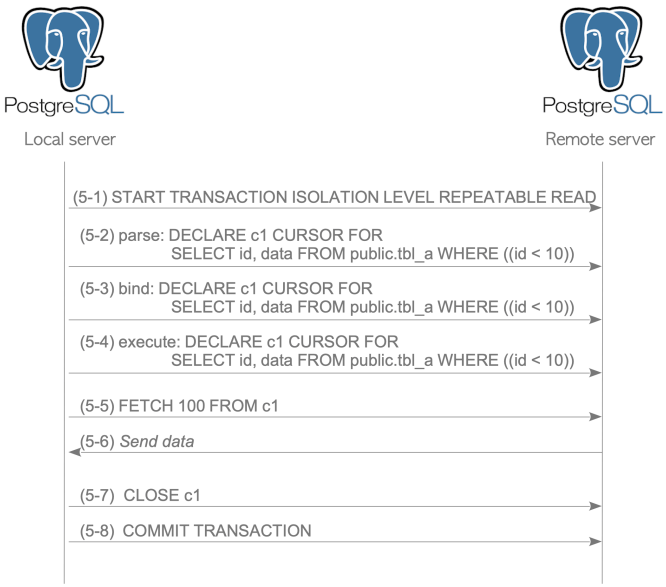

Figure 4.5. Typical sequence of SQL statements to execute a SELECT query in postgres_fdw.

- (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.

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  parse <unnamed>: DECLARE c1 CURSOR FOR SELECT id, data FROM public.tbl_a WHERE ((id < 10))
LOG:  bind <unnamed>: DECLARE c1 CURSOR FOR SELECT id, data FROM public.tbl_a WHERE ((id < 10))
LOG:  execute <unnamed>: DECLARE c1 CURSOR FOR SELECT id, data FROM public.tbl_a WHERE ((id < 10))
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

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.*

4.2. 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.
15	• Allow postgres_fdw to push down CASE expressions. • Add server variable postgres_fdw.application_name to control the application name of postgres_fdw connections. • Allow parallel commit on postgres_fdw servers.
16	• Allow postgres_fdw to do aborts in parallel. • Make ANALYZE on foreign postgres_fdw tables more efficient. • Restrict shipment of "reg*" type constants in postgres_fdw to those referencing built-in objects or extensions marked as shippable. • Have postgres_fdw and dblink handle interrupts during connection establishment.
17	• Allow pushdown of EXISTS and IN subqueries to postgres_fdw foreign servers.

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.

Note: 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
```

4.2.1. 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.

4.2.1.1. 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 (Figure 4.6).

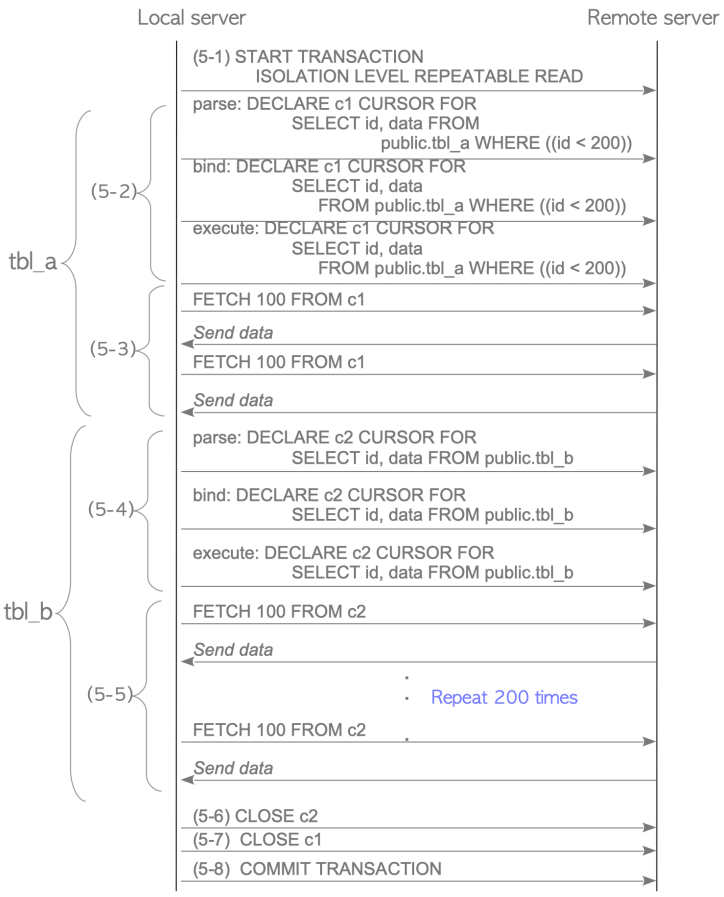

Figure 4.6. Sequence of SQL statements to execute the Multi-Table Query in versions 9.5 or earlier.

- (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.

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  parse <unnamed>: DECLARE c1 CURSOR FOR
      SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  bind <unnamed>: DECLARE c1 CURSOR FOR
      SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  execute <unnamed>: DECLARE c1 CURSOR FOR
      SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: FETCH 100 FROM c1
LOG:  parse <unnamed>: DECLARE c2 CURSOR FOR
      SELECT id, data FROM public.tbl_b
LOG:  bind <unnamed>: DECLARE c2 CURSOR FOR
      SELECT id, data FROM public.tbl_b
LOG:  execute <unnamed>: DECLARE c2 CURSOR FOR
      SELECT id, data FROM public.tbl_b
LOG:  statement: FETCH 100 FROM c2
LOG:  statement: FETCH 100 FROM c2
LOG:  statement: FETCH 100 FROM c2
LOG:  statement: FETCH 100 FROM c2

... snip

LOG:  statement: FETCH 100 FROM c2
LOG:  statement: FETCH 100 FROM c2
LOG:  statement: FETCH 100 FROM c2
LOG:  statement: FETCH 100 FROM c2
LOG:  statement: CLOSE c2
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

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.

4.2.1.2. 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.

```
localdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND a.id < 200;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Foreign Scan  (cost=134.35..244.45 rows=80 width=16)
   Relations: (public.tbl_a a) INNER JOIN (public.tbl_b b)
(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 (Figure 4.7).

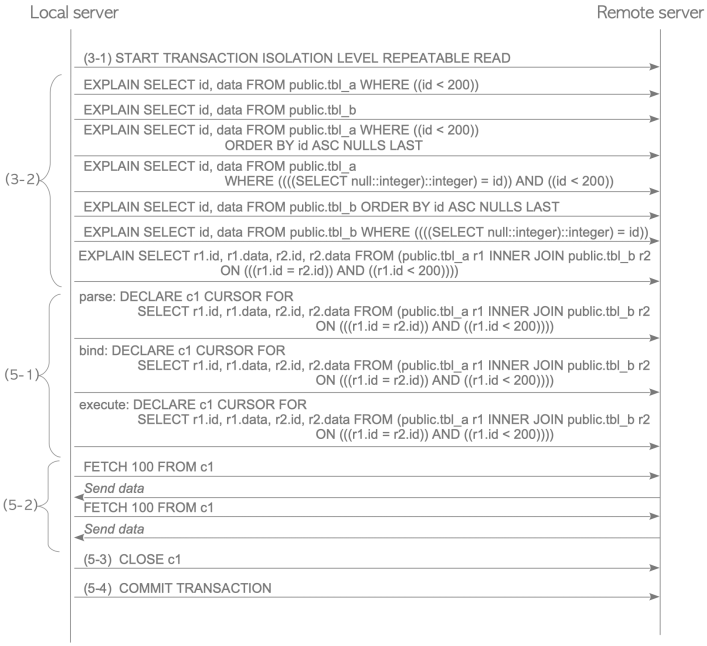

Figure 4.7. Sequence of SQL statements to execute the remote-join operation in versions 9.6 or later.

- (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.

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  statement: EXPLAIN SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  statement: EXPLAIN SELECT id, data FROM public.tbl_b
LOG:  statement: EXPLAIN SELECT id, data FROM public.tbl_a WHERE ((id < 200)) ORDER BY id ASC NULLS
LAST
LOG:  statement: EXPLAIN SELECT id, data FROM public.tbl_a WHERE ((((SELECT null::integer)::integer) =
id)) AND ((id < 200))
LOG:  statement: EXPLAIN SELECT id, data FROM public.tbl_b ORDER BY id ASC NULLS LAST
LOG:  statement: EXPLAIN SELECT id, data FROM public.tbl_b WHERE ((((SELECT null::integer)::integer) =
id))
LOG:  statement: 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))))
LOG:  parse: DECLARE c1 CURSOR FOR
       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))))
LOG:  bind: DECLARE c1 CURSOR FOR
       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))))
LOG:  execute: DECLARE c1 CURSOR FOR
       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))))
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

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.

4.2.2. Sort Operations

4.2.2.1. 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.

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  parse <unnamed>: DECLARE c1 CURSOR FOR
      SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  bind <unnamed>: DECLARE c1 CURSOR FOR
      SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  execute <unnamed>: DECLARE c1 CURSOR FOR
      SELECT id, data FROM public.tbl_a WHERE ((id < 200))
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

4.2.2.2. Versions 9.6 or later

The 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;
2                               QUERY PLAN
3 -----------------------------------------------------------------
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
  ```

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  parse <unnamed>: DECLARE c1 CURSOR FOR
       SELECT id, data FROM public.tbl_a WHERE ((id < 200)) ORDER BY id ASC NULLS LAST
LOG:  bind <unnamed>: DECLARE c1 CURSOR FOR
       SELECT id, data FROM public.tbl_a WHERE ((id < 200)) ORDER BY id ASC NULLS LAST
LOG:  execute <unnamed>: DECLARE c1 CURSOR FOR
       SELECT id, data FROM public.tbl_a WHERE ((id < 200)) ORDER BY id ASC NULLS LAST
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

This improvement has reduced the workload of the local server.

4.2.3. Aggregate Functions

4.2.3.1. 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;
2                                 QUERY PLAN
3 ------------------------------------------------------------------------
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.

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  parse <unnamed>: DECLARE c1 CURSOR FOR
      SELECT data FROM public.tbl_a WHERE ((id < 200))
LOG:  bind <unnamed>: DECLARE c1 CURSOR FOR
      SELECT data FROM public.tbl_a WHERE ((id < 200))
LOG:  execute <unnamed>: DECLARE c1 CURSOR FOR
      SELECT data FROM public.tbl_a WHERE ((id < 200))
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

This process is costly because sending a large number of rows consumes heavy network traffic and takes a long time.

4.2.3.2. Versions 10 or later

The 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;
2                     QUERY PLAN
3 -----------------------------------------------------
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))
  ```

Actual log of the remote server

```
LOG:  statement: START TRANSACTION ISOLATION LEVEL REPEATABLE READ
LOG:  parse <unnamed>: DECLARE c1 CURSOR FOR
       SELECT avg(data) FROM public.tbl_a WHERE ((id < 200))
LOG:  bind <unnamed>: DECLARE c1 CURSOR FOR
       SELECT avg(data) FROM public.tbl_a WHERE ((id < 200))
LOG:  execute <unnamed>: DECLARE c1 CURSOR FOR
       SELECT avg(data) FROM public.tbl_a WHERE ((id < 200))
LOG:  statement: FETCH 100 FROM c1
LOG:  statement: CLOSE c1
LOG:  statement: COMMIT TRANSACTION
```

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.

5. CONCURRENCY CONTROL

Concurrency 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.)

Chapter Contents

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*1	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.

Note

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

At the start of every transaction, the transaction manager assigns a unique identifier known as a transaction ID (txid). PostgreSQL's txid is a 32-bit unsigned integer, with a maximum value of approximately 4.2 billion (thousand millions).

Executing the built-in txid_current() function after a transaction begins returns the current txid, as shown below:

```
testdb=# BEGIN;
BEGIN
testdb=# SELECT txid_current();
 txid_current
--------------
          100
(1 row)
```

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* (Figure 5.1 a)).

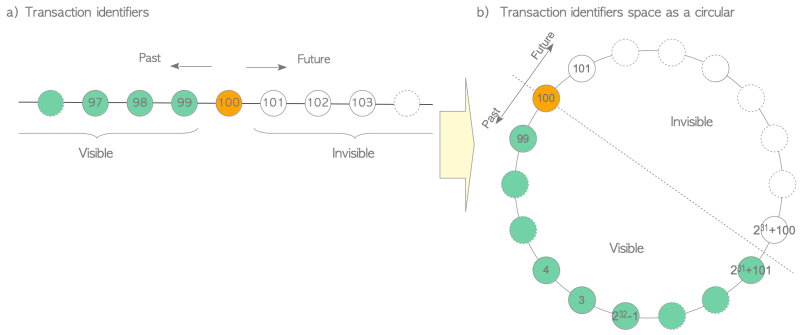

Figure 5.1. Transaction ids in PostgreSQL.

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' (Figure 5.1 b).

Note that the so-called *txid wraparound problem* is described in Section 5.10.1.

Info

Note that BEGIN command does not be assigned a txid.

In PostgreSQL, when the first command is executed after a BEGIN command executed, a tixd 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 (Figure 5.2).

Figure 5.2. Tuple structure.

Info

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; 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.

HeapTupleHeaderData

```
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 directly!) */
        int32           datum_typmod;        /* -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.

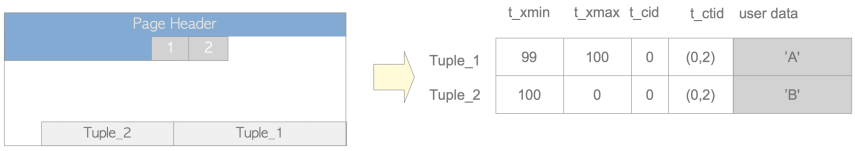

Figure 5.3. Representation of tuples.

5.3.1. Insertion

With the insertion operation, a new tuple is inserted directly into a page of the target table (Figure 5.4).

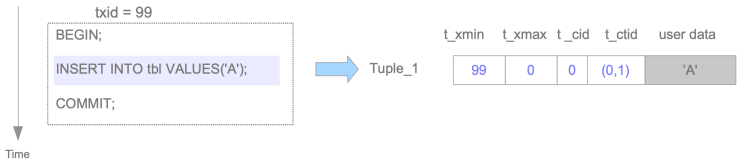

Figure 5.4. Tuple insertion.

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 the 'pageinspect' extension, 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 (Figure 5.5).

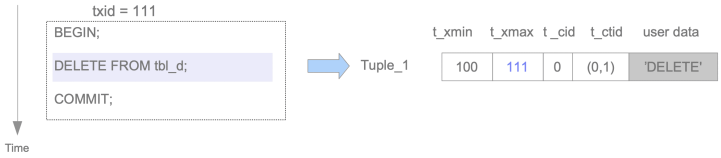

Figure 5.5. Tuple deletion.

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 (Figure 5.6).

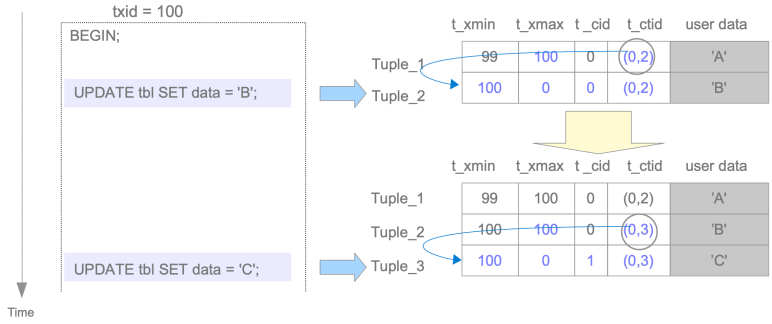

Figure 5.6. Update the row twice.

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 txid 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 **Free Space Map (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.

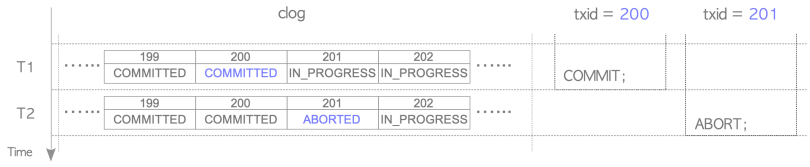

Figure 5.7. How the clog operates.

T1:*txid 200 commits; the status of txid 200 is changed from IN_PROGRESS to COMMITTED.*
T2:*txid 201 aborts;the status of txid 201 is changed from IN_PROGRESS to ABORTED.*

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 ℹ'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 ℹ below.

ℹ 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:

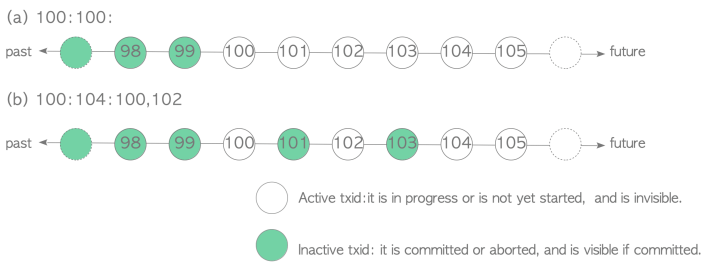

Figure 5.8. Examples of transaction snapshot representation.

The first example is *'100:100:'*. This snapshot means the following (Figure 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 (Figure 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 the specific scenario illustrated in Figure 5.9.

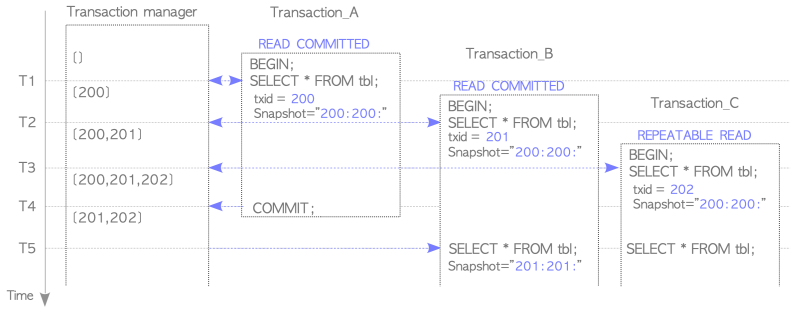

Figure 5.9. Transaction manager and transactions.

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.
  In contrast, 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 ⇒ 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
Rule 2:             IF t_xmax = INVALID THEN
                          RETURN 'Visible'
Rule 3:             ELSE  /* this tuple has been deleted or updated  */
                          /* by the current transaction itself.      */
                           RETURN 'Invisible'
                    END IF
Rule 4:       ELSE   /* t_xmin &ne; 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 Status(t_xmin) = IN_PROGRESS ∧ t_xmin = current_txid ∧ t_xmax = INVAILD ⇒ Visible
- **Rule 3:** If Status(t_xmin) = IN_PROGRESS ∧ t_xmin = current_txid ∧ t_xmax ≠ INVAILD ⇒ Invisible
- **Rule 4:** If Status(t_xmin) = IN_PROGRESS ∧ t_xmin ≠ current_txid ⇒ Invisible

5.6.3. Status of t_xmin is COMMITTED

A tuple whose t_xmin status is COMMITTED is *visible* (Rules 6,8, and 9), except under three conditions.

```
 /* t_xmin status == COMMITTED */
       IF t_xmin status is 'COMMITTED' THEN
Rule 5:       IF t_xmin is 'active' in the obtained transaction snapshot THEN
                    RETURN 'Invisible'
Rule 6:       ELSE IF t_xmax = INVALID OR status of t_xmax is 'ABORTED' THEN
                    RETURN 'Visible'
              ELSE IF t_xmax status is 'IN_PROGRESS' THEN
Rule 7:             IF t_xmax =  current_txid THEN
                          RETURN 'Invisible'
Rule 8:             ELSE  /* t_xmax != current_txid */
                          RETURN 'Visible'
                    END IF
              ELSE IF t_xmax status is 'COMMITTED' THEN
Rule 9:             IF t_xmax is 'active' in the obtained transaction snapshot THEN
```

```
                        RETURN 'Visible'
Rule 10:            ELSE
                        RETURN 'Invisible'
                    END IF
                END IF
            END IF
```

Rule 6 is obvious because t_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_xmin is *active* in the obtained transaction snapshot (Rule 5). Under this condition, this tuple is *invisible* because t_xmin should be treated as in progress.

The second exception condition is that t_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_xmax is IN_PROGRESS and t_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_xmax is COMMITTED and t_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_xmax is COMMITTED but t_xmax is active in the obtained transaction snapshot (Rule 9), the tuple is *visible* because t_xmax should be treated as in progress.

- **Rule 5:** If Status(t_xmin) = COMMITTED $\wedge$ Snapshot(t_xmin) = active $\Rightarrow$ Invisible
- **Rule 6:** If Status(t_xmin) = COMMITTED $\wedge$ (t_xmax = INVALID $\vee$ Status(t_xmax) = ABORTED) $\Rightarrow$ Visible
- **Rule 7:** If Status(t_xmin) = COMMITTED $\wedge$ Status(t_xmax) = IN_PROGRESS $\wedge$ t_xmax = current_txid $\Rightarrow$ Invisible
- **Rule 8:** If Status(t_xmin) = COMMITTED $\wedge$ Status(t_xmax) = IN_PROGRESS $\wedge$ t_xmax $\neq$ current_txid $\Rightarrow$ Visible
- **Rule 9:** If Status(t_xmin) = COMMITTED $\wedge$ Status(t_xmax) = COMMITTED $\wedge$ Snapshot(t_xmax) = active $\Rightarrow$ Visible
- **Rule 10:** If Status(t_xmin) = COMMITTED $\wedge$ Status(t_xmax) = COMMITTED $\wedge$ Snapshot(t_xmax) $\neq$ active $\Rightarrow$ 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.

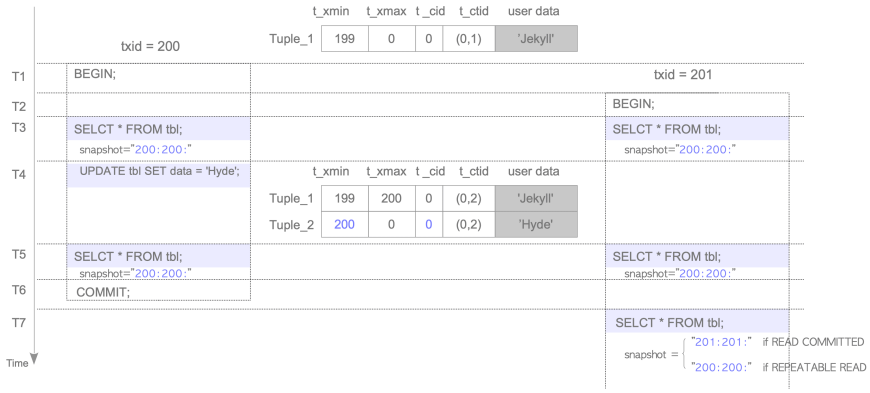

Figure 5.10. Scenario to describe visibility check.

In the scenario shown in Figure 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) ⇒ Status(t_xmin:199) = COMMITTED ∧ t_xmax = INVALID ⇒ 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 ∧ Status(t_xmax:200) = IN_PROGRESS ∧ t_xmax:200 = current_txid:200 ⇒ Invisible
- Rule2(Tuple_2): Status(t_xmin:200) = IN_PROGRESS ∧ t_xmin:200 = current_txid:200 ∧ t_xmax = INVAILD ⇒ 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): Status(t_xmin:200) = COMMITTED $\wedge$ t_xmax = INVALID $\Rightarrow$ 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): Status(t_xmin:199) = COMMITTED $\wedge$ Status(t_xmax:200) = COMMITTED $\wedge$ Snapshot(t_xmax:200) = active $\Rightarrow$ Visible
- Rule5(Tuple_2): Status(t_xmin:200) = COMMITTED $\wedge$ Snapshot(t_xmin:200) = active $\Rightarrow$ Invisible

```
testdb=# -- txid 201 (REPEATABLE READ)
testdb=# SELECT * FROM tbl;
  name
--------
 Jekyll
(1 row)
```

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 blow:

```
#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_informask 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 ∧ Snapshot(t_xmin:100) = active ⇒ Invisible

```
testdb=# -- Tx_A: txid 100
testdb=# START TRANSACTION
testdb-#  ISOLATION LEVEL READ COMMITTED;
START TRANSACTION
testdb=# SELECT txid_current();
 txid_current
--------------
          100
(1 row)

testdb=# INSERT INTO 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
               */
(3)            IF the target row is 'being updated' THEN
(4)             WAIT for the termination of the transaction that updated the target row

(5)                 IF (the status of the terminated transaction is COMMITTED)
                  AND (the isolation level of this transaction is REPEATABLE READ or SERIALIZABLE)
THEN
(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 Figure 5.11.

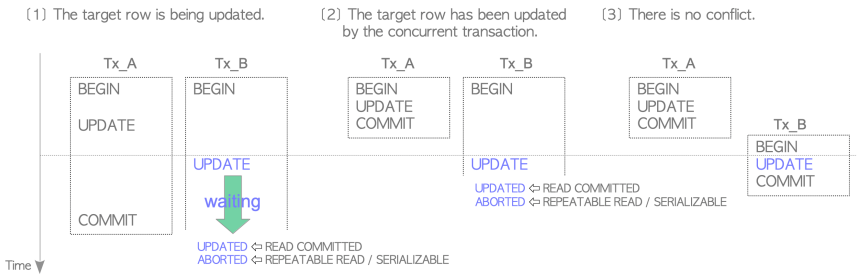

Figure 5.11. Three internal blocks in ExecUpdate.

- [1] The target row is being updated (Figure 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 (Figure 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 (Figure 5.11[3])
  When there is no conflict, the current transaction can update the target row.

First-updater-win / First-commiter-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.

5.8.2.1. 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=#
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).

5.8.2.2. 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=#
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).

5.8.2.3. 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=#
testdb=#
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 the original paper : "Serializable Snapshot Isolation in PostgreSQL"

In the following, the technical terms shown in below are used without definitions.

- precedence graph (also known as dependency graph and serialization graph)
- serialization anomalies (e.g. Write-Skew)

If you are unfamiliar with these terms, see the references:

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

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 Figure 5.12(2). Thus, this schedule has a serialization anomaly, Write-Skew.

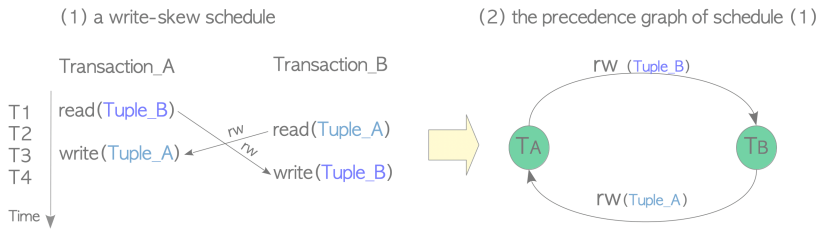

Figure 5.12. Write-Skew schedule and its precedence graph.

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.

Note

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.

For detailed information, 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 (Figure 5.13).

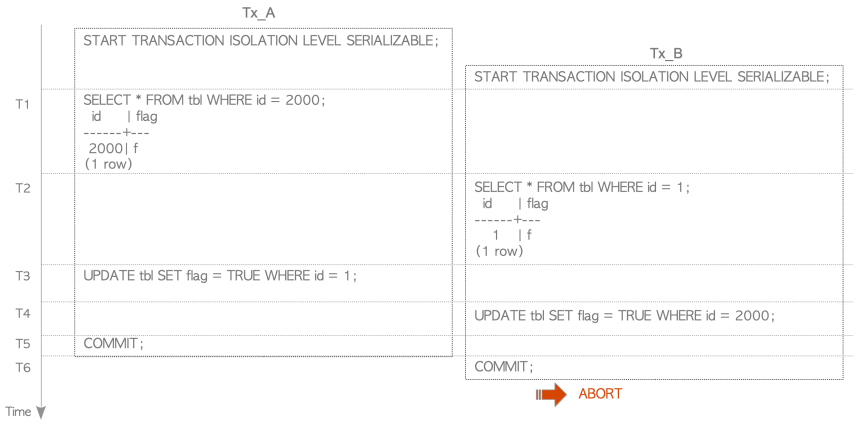

Figure 5.13. Write-Skew scenario.

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 Figure 5.14.

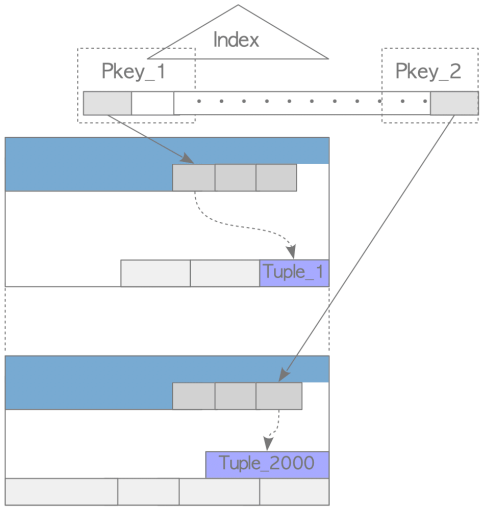

Figure 5.14. Relationship between the index and table in the scenario shown in Figure 5.13.

- **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.

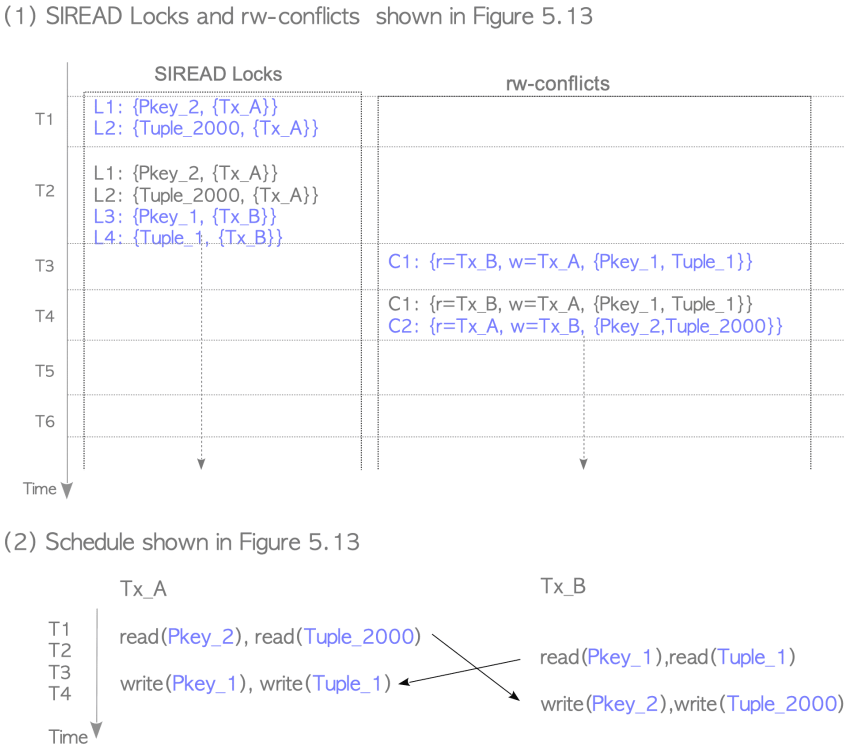

Figure 5.15. SIREAD locks and rw-conflicts, and schedule of the scenario shown in Figure 5.13.

- **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.

5.9.3.1. Other Scenarios

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 (Figure 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 (Figure 5.16(2)).

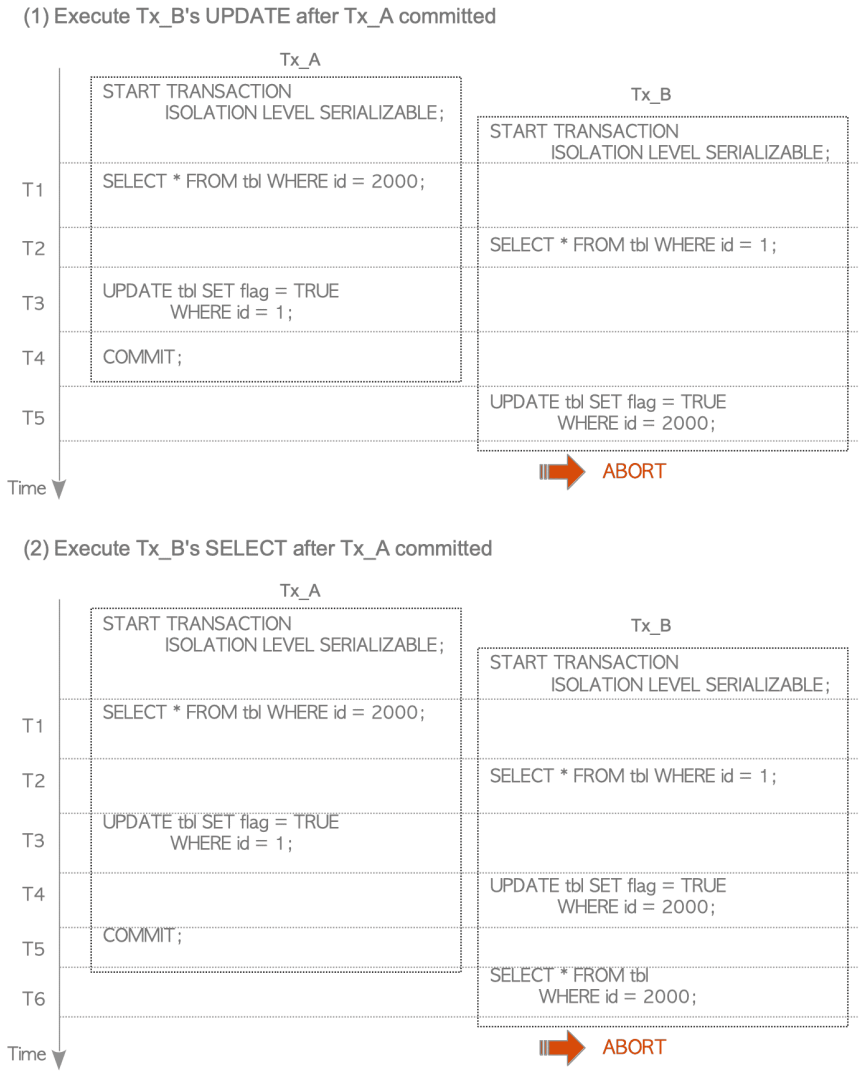

Figure 5.16. Other Write-Skew scenarios.

> **Info**
>
> 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.

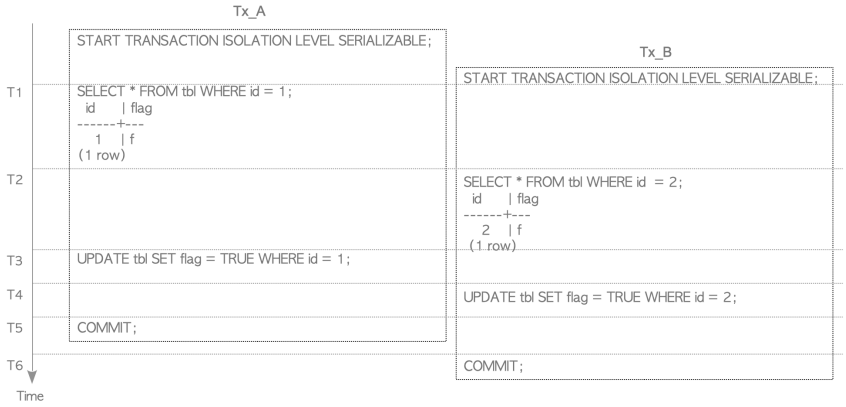

Figure 5.17. Scenario where false-positive serialization anomaly occurs.

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).

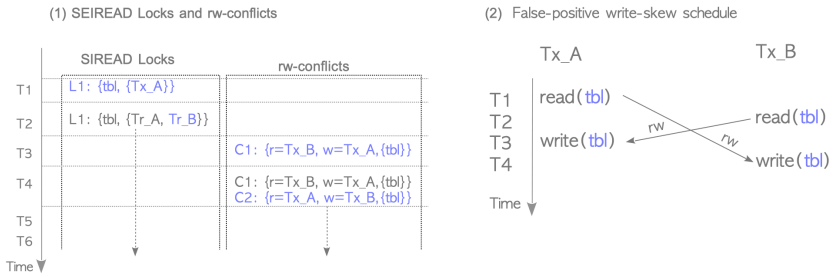

Figure 5.18. False-positive anomaly (1) - Using sequential scan.

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.

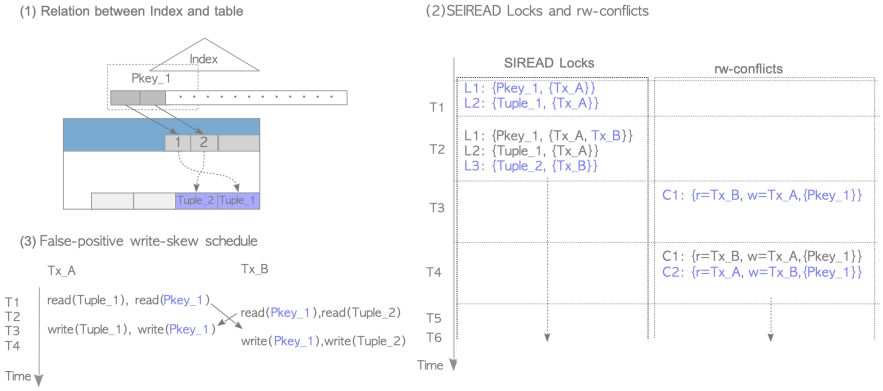

Figure 5.19. False-positive anomaly (2) - Index scan using the same index page.

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 Section 5.3.2 and Section 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

5.10.1.1. Transaction 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* (Figure 5.20).

This is the so called *transaction wraparound problem* in PostgreSQL.

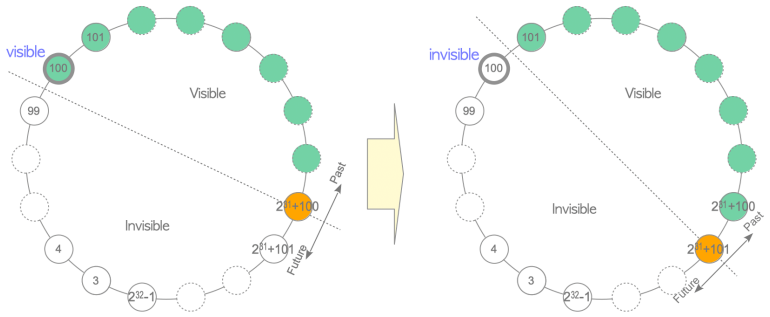

Figure 5.20. Wraparound problem.

5.10.1.2. Freeze Processing

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 Figure 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 (Figure 5.21 b).

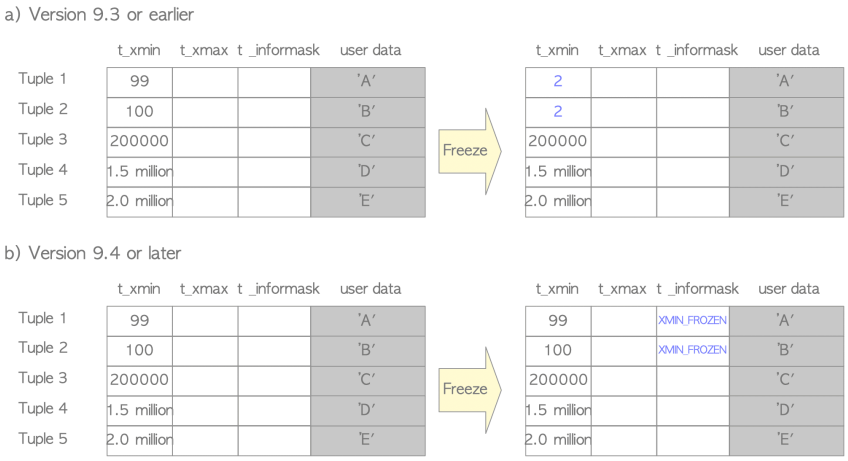

Figure 5.21. Freeze process.

6. VACUUM PROCESSING

Vacuum 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.

Chapter Contents

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.

Code 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.

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:

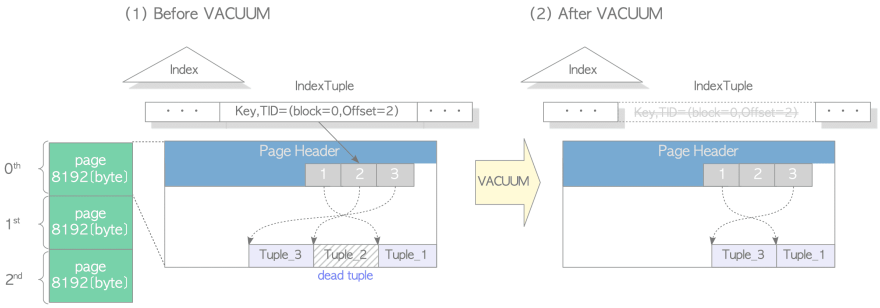

Figure 6.1. Removing a dead tuple.

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 (Figure 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 (Figure 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.

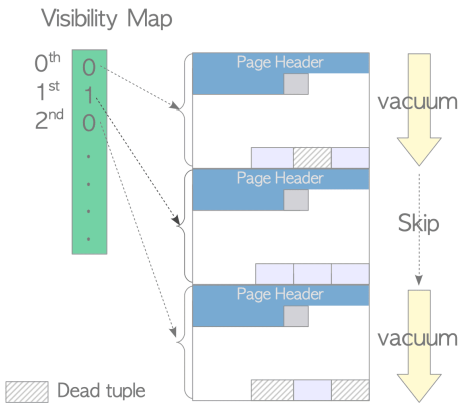

Figure 6.2. How the VM is used.

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.

```
$ 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.

> **Note**
>
> 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 $\mathbf{freezeLimit\_txid}$ and freezes tuples whose t_xmin is less than the freezeLimit_txid.

The freezeLimit_txid is defined as follows:

$$\mathrm{freezeLimit\_txid} = (\mathrm{OldestXmin} - \mathrm{vacuum\_freeze\_min\_age})$$

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.

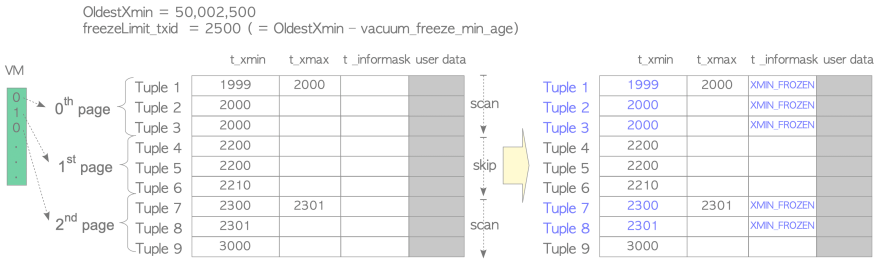

Figure 6.3. Freezing tuples in lazy mode.

- 0th 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.
- 1st page:
  This page is skipped by referring to the VM.
- 2nd 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:

$$\text{pg\_database.datfrozenxid} < (\text{OldestXmin} - \text{vacuum\_freeze\_table\_age})$$

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.)

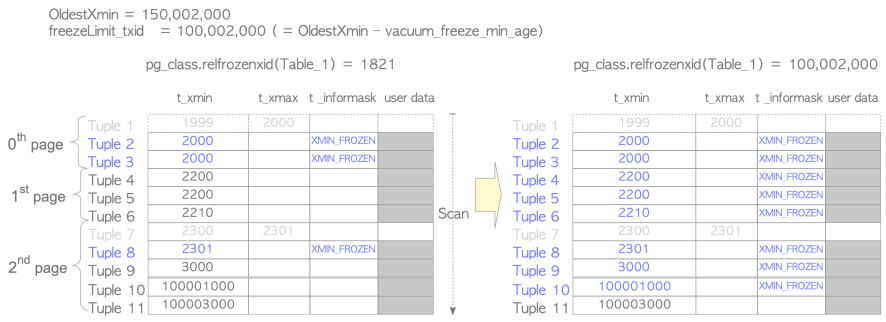

Figure 6.4. Freezing old tuples in eager mode. (versions 9.5 or earlier)

- 0th page:
  Tuple_2 and Tuple_3 have been checked even though all tuples have been frozen.
- 1st 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.
- 2nd 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 (Figure 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 (Figure 6.5(2)).

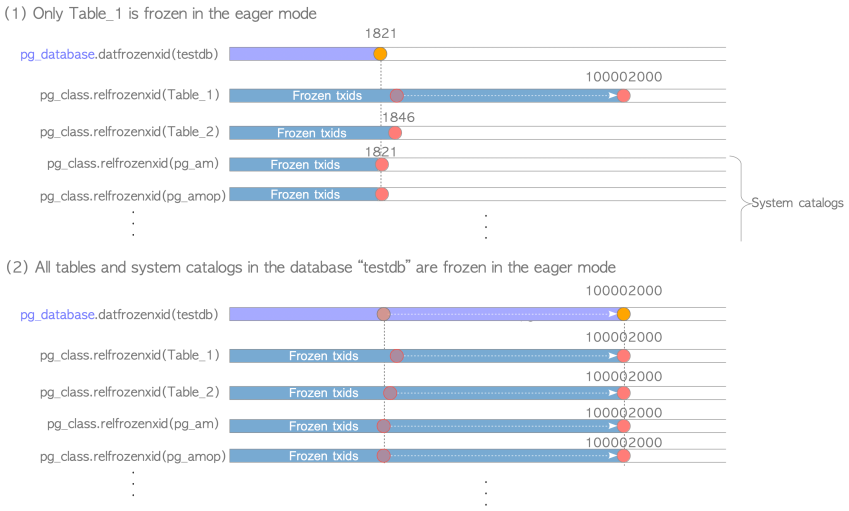

Figure 6.5. Relationship between pg_database.datfrozenxid and pg_class.relfrozenxid(s).

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.datfrozenxld 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.relkind 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)
```

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, OldesXmin 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 versions 9.5 or earlier 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.

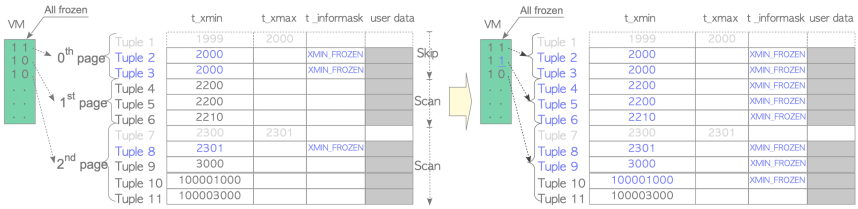

Figure 6.6. Freezing old tuples in eager mode (versions 9.6 or later).

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.

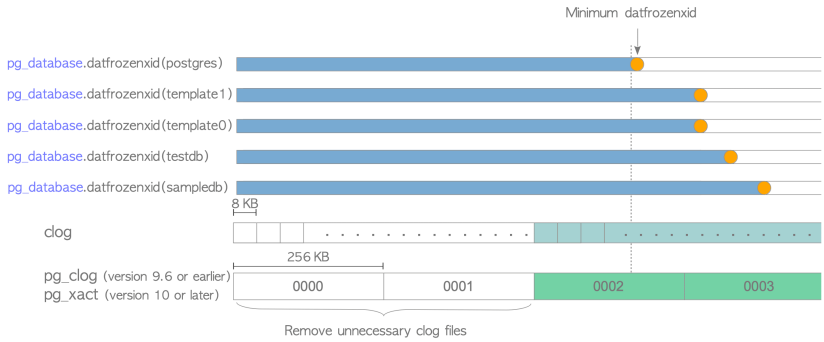

Figure 6.7. Removing unnecessary clog files and pages.

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_works).

The autovacuum workers invoked by the autovacuum daemon 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:

   $$\mathrm{relfrozenxid} + \mathrm{autovacuum\_freeze\_max\_age}$$

   where **relfrozenxid** is the relfrozenxid value of the target table that is defined 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:

   $$\mathrm{autovacuum\_vacuum\_threshold} + \mathrm{autovacuum\_vacuum\_scale\_factor} \times \mathrm{reltuples}$$

   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, because

   $$2100 > 50 + 0.2 \times 10000.$$

3. The number of inserted tuples in the target table is greater than the following expression:

   $$\mathrm{autovacuum\_vacuum\_insert\_threshold} + \mathrm{autovacuum\_vacuum\_insert\_scale\_factor} \times \mathrm{reltuples}$$

   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, because

   $$3010 > 1000 + 0.2 \times 10000.$$

   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.

$$\mathrm{mod\_since\_analyze} > \mathrm{autovacuum\_analyze\_threshold} + \mathrm{autovacuum\_analyze\_scale\_factor} \times \mathrm{reltuples}$$

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, because

$$1100 > 50 + 0.1 \times 10000.$$

Info

The relation_needs_vacanalyze() function 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, its default value (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, the following example shows how to set the new value of autovacuum_vacuum_scale_factor for the pgbench_accounts table.

```
postgres=# ALTER TABLE pgbench_accounts SET (autovacuum_vacuum_scale_factor = 0.05);
ALTER TABLE
```

To ensure that Autovacuum runs for target tables independently of the number of their tuples, specific settings can be applied.

For instance, if Autovacuum processing is required whenever the number of dead tuples reaches 10,000, the following storage parameters can be configured for the table. With these settings, the Autovacuum process triggers vacuuming each time the threshold of 10,000 dead tuples is reached:

```
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:

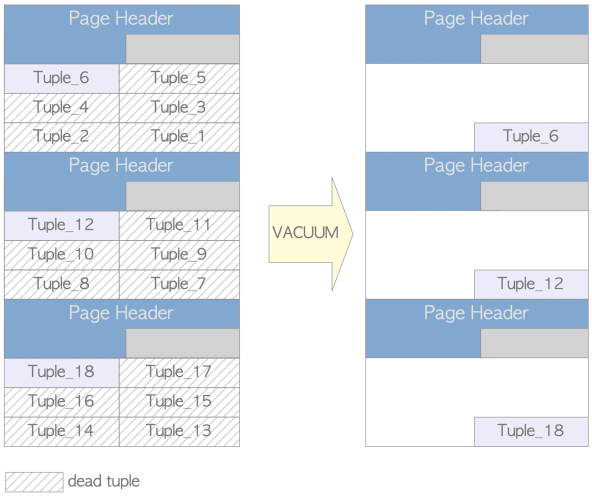

Figure 6.8. An example showing the disadvantages of (concurrent) VACUUM.

```
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.

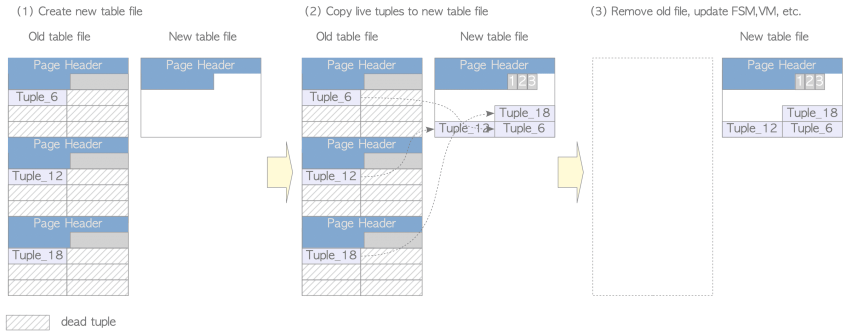

Figure 6.9. Outline of Full VACUUM mode.

- [1] Create new table file: Figure 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: Figure 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: Figure 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 to Use VACUUM FULL.

There is no universal answer to the question of when to execute VACUUM FULL. However, the pg_freespacemap extension can provide useful insights.

The following query calculates the average free space ratio for a specific table:

```
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)
```

The result indicates minimal free space (1.21%).

If most tuples are deleted and the VACUUM command is executed, many pages may become empty (e.g., 86.97% free space), but the total number of pages remains unchanged, indicating that 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)
```

To inspect the free space ratio for each page of a table, the following query can be used:

```
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
....
```

In such a situation, running VACUUM FULL compacts the table file:

```
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)
```

By compacting the table, VACUUM FULL reduces the physical size of the table file, making it more efficient.

7. HOT AND INDEX-ONLY SCANS

This chapter describes two features related to the index scan: the heap only tuple and index-only scans.

Chapter Contents

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 by the corresponding index tuple, whose key is 1000 and whose tid is '(5, 1)'. See Figure 7.1(a).

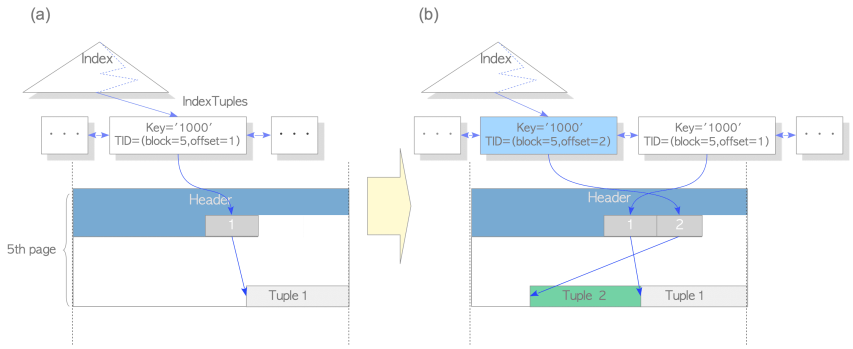

Figure 7.1. Update a row without HOT

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 Figure 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 Figures. 7.2 and 7.3.

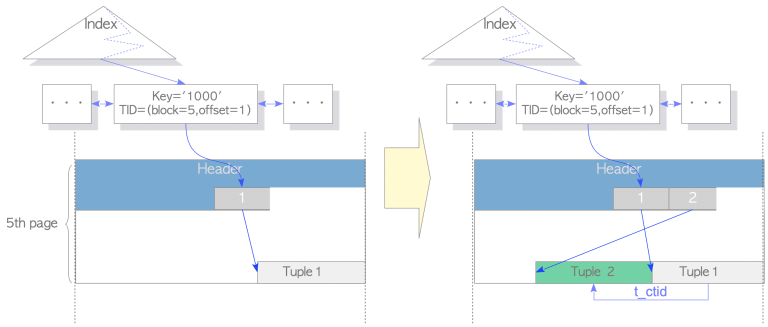

Figure 7.2. Update a row with HOT

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.

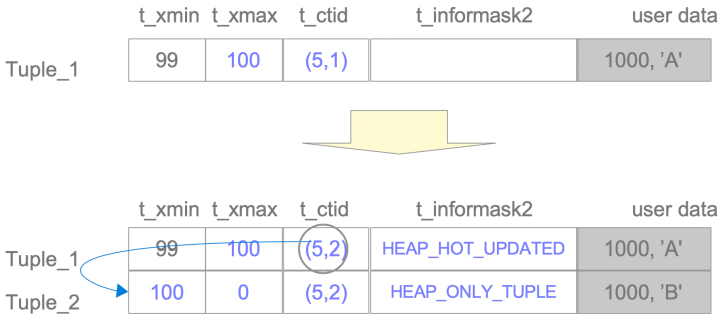

Figure 7.3. HEAP_HOT_UPDATED and HEAP_ONLY_TUPLE bits

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 Figure 7.4(a).

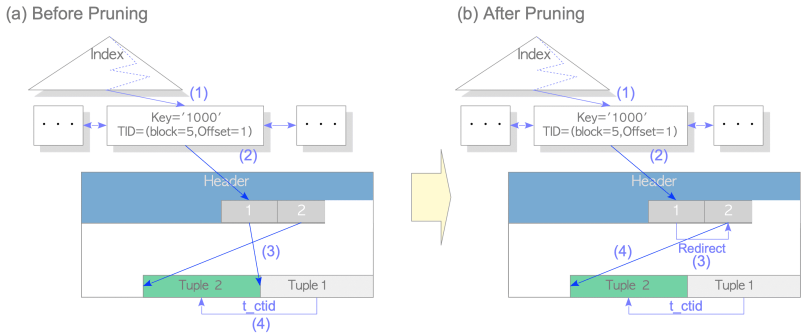

Figure 7.4. Pruning of the line pointers

1. Find the index tuple that points to the target tuple.
2. Access the line pointer [1] that is pointed 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 Figure 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**. Figure 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 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 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**. Figure 7.5 depicts the defragmentation by HOT.

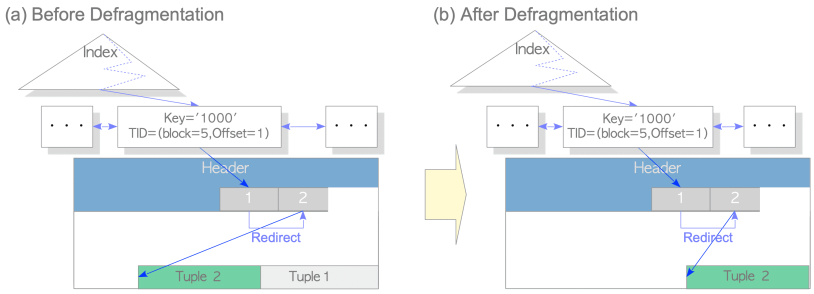

Figure 7.5. Defragmentation of the dead tuples

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.

The Cases in which HOT is not available

To better understand HOT performance, we will describe cases where 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 Figure 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).

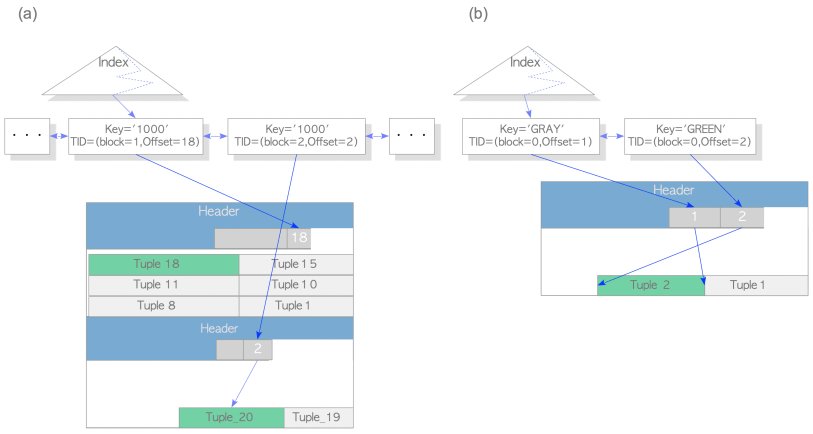

Fig. 7.6. The Cases in which HOT is not available

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 Figure 7.7.

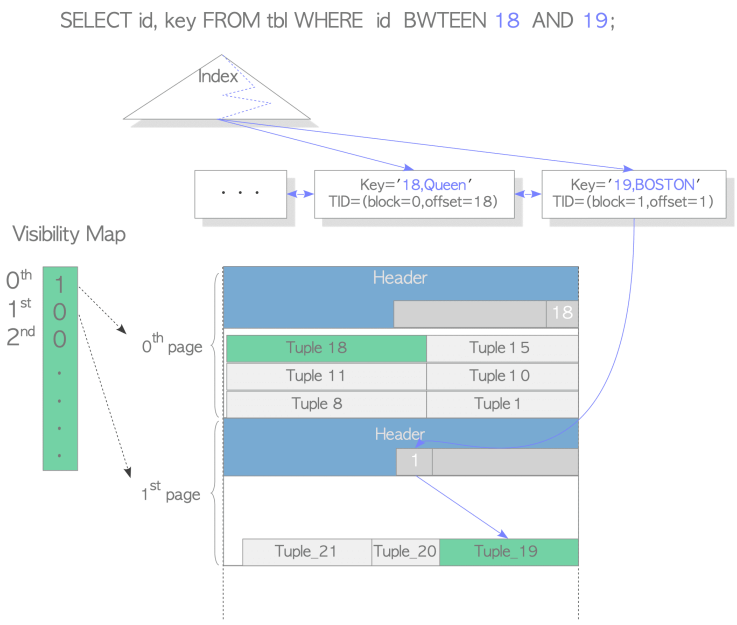

Figure 7.7. How Index-Only Scans performs

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 and Local buffer
- Flushing of dirty pages

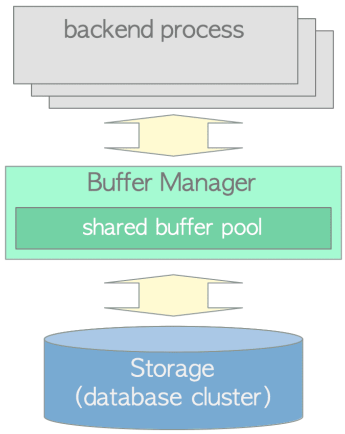

Figure 8.1. Relations between buffer manager, storage, and backend processes.

Chapter Contents

8.1. OVERVIEW

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

8.1.1. Buffer Manager Structure

The PostgreSQL buffer manager comprises a buffer table, buffer descriptors, and buffer pool, which are described in the next section.

The **buffer pool** layer stores data file pages, such as tables and indexes, as well as freespace maps and visibility maps.

The buffer pool is an array, where each slot stores one page of a data file. The Indices of a buffer pool array are referred to as **buffer_id**s.

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

8.1.2. Buffer Tag

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

The `buffer_tag` has five values:

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

› buffer_tag

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 (Figure 8.2).

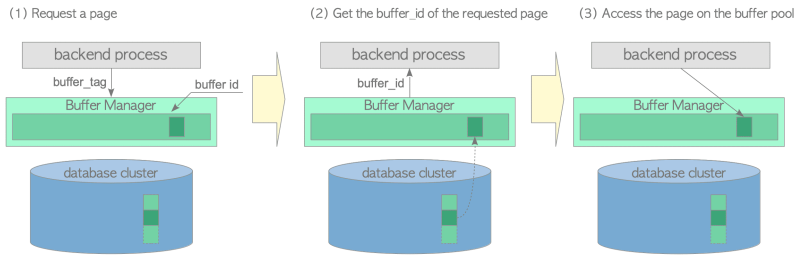

Figure 8.2. How a backend reads a page from the buffer manager.

- (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.

Historical Information

Until version 7.4, PostgreSQL used a simple LRU algorithm.

Although PostgreSQL implemented ARC (Adaptive Replacement Cache) in version 8.0, it might have been possible to violate IBM's patent.

As a result, the PostgreSQL community tentatively changed the replacement algorithm to 2Q, and then PostgreSQL supported clock sweep in version 8.1.

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 article 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' (Figure 8.3):

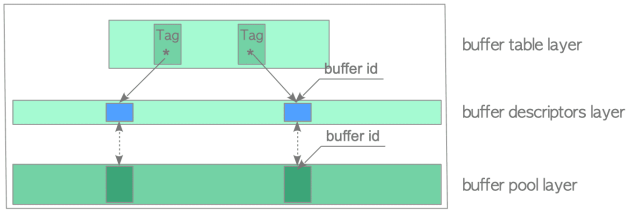

Figure 8.3. Buffer manager's three-layer structure.

- **Buffer pool**: An array that stores data file pages. Each slot in the array is referred to as **buffer_id**s.
- **Buffer descriptors**: An array of buffer descriptors. Each descriptor has a one-to-one correspondence to a buffer pool slot and holds the metadata of the stored page in the corresponding slot.
  Note that the term 'buffer descriptors layer' has been adopted for convenience and is only used in this document.
- **Buffer table**: A hash table that stores the relations between the buffer_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 (Figure 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 Figure 8.4.

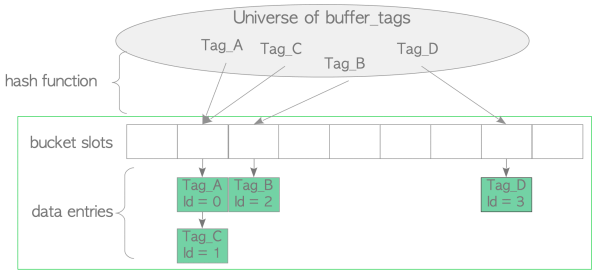

Figure 8.4. Buffer table.

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

Hash Function

The hash function is a composite function of calc_bucket() and hash().

The following is its representation as a pseudo-function.

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

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

8.2.2. Buffer Descriptor

The structure of buffer descriptors was improved in version 9.6 This section first explains buffer descriptors in versions 9.5 or earlier, followed by a discussion of the changes introduced in versions 9.6 or later.

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:

BufferDesc

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

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

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

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

BufferDesc

```
/*
 * 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.

Info

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

8.2.2.3. Descriptor States

To simplify the following descriptions, we define three descriptor states as follows:

- **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** (Figure 8.5).

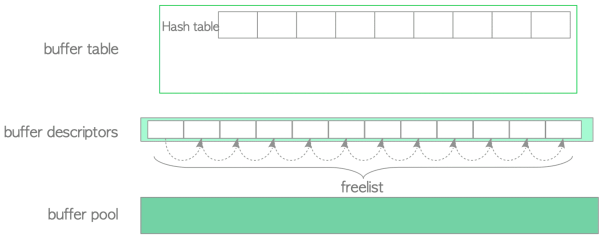

Figure 8.5. Buffer manager initial state.

> **Note**
>
> 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 (FSM)*, which are described in Section 5.3.4, serve as the same purpose as the freelists in Oracle.

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.

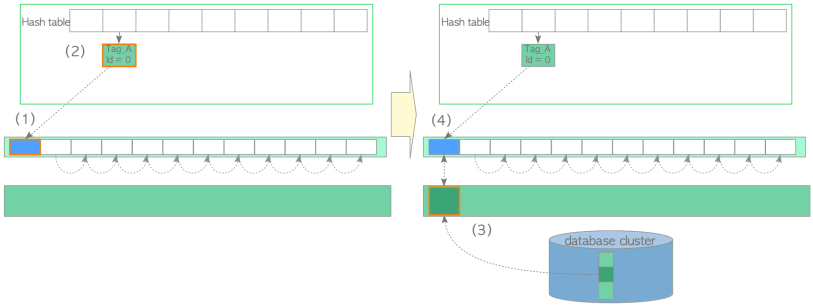

Figure 8.6. Loading the first page.

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

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

> **Why empty descriptors comprise the freelist?**
>
> The freelist is created to allow for the immediate retrieval of the first descriptor. This is a usual practice for dynamic memory resource allocation. For more information, refer to this description.

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

8.2.4. Buffer Pool

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

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

8.3. BUFFER MANAGER LOCKS

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

Note

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

8.3.1. Buffer Table Locks

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

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

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

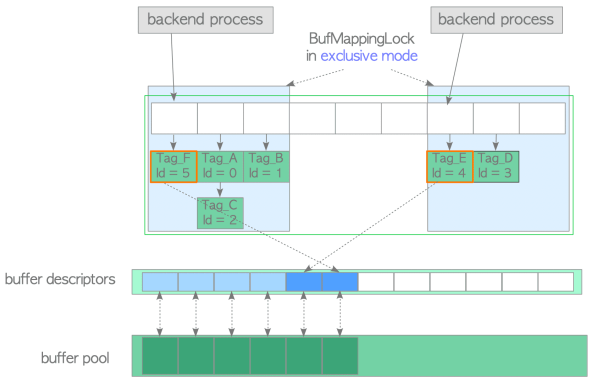

Figure 8.7. Two processes simultaneously acquire the respective partitions of BufMappingLock in exclusive mode to insert new data entries.

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

Histrical Information

The BufMappingLock was introduced in version 8.1. Until version 9.4, the BufMappingLock was split into 16 separate locks by default.

Before the introduction of BufMappingLock, PostgreSQL used BufMgrLock, a giant lock mechanism that had to be acquired whenever the shared buffer was accessed, resulting in poor concurrency.

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 were improved. The io_in_progress_lock and spin lock (buf_hdr_lock) were removed. Instead of using these locks, versions 9.6 or later use CPU atomic operations to inspect and change their values.

Atomic Operations

An atomic operation is an operation that executes without interruption, ensuring it completes as a single, indivisible unit. These operations are commonly used for tasks such as updating counters without locks or implementing lock-free data structures.

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.

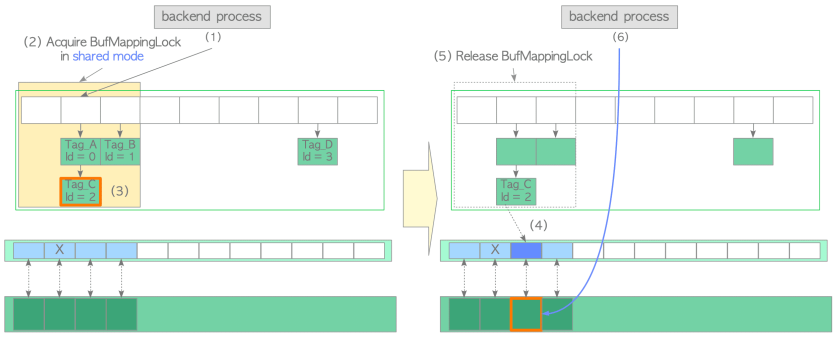

Figure 8.8. Accessing a page stored in the buffer pool.

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 an 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.

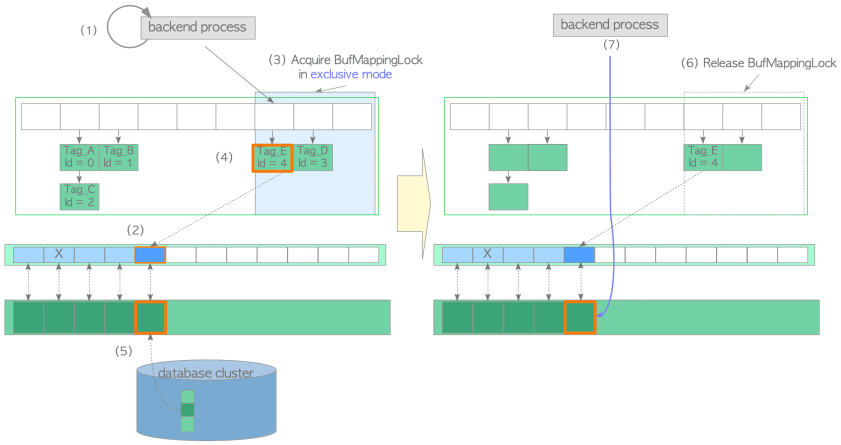

Figure 8.9. Loading a page from storage to an empty slot.

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 Section 8.4.4.
- (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.

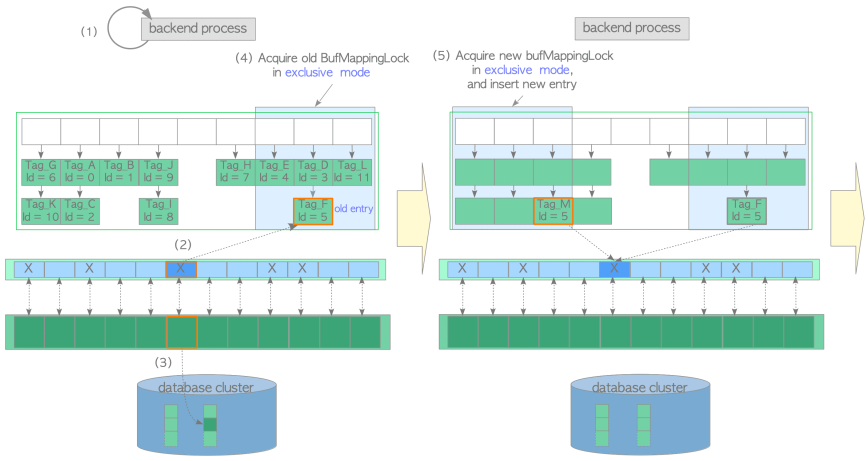

Figure 8.10. Loading a page from storage to a victim buffer pool slot.

- (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.

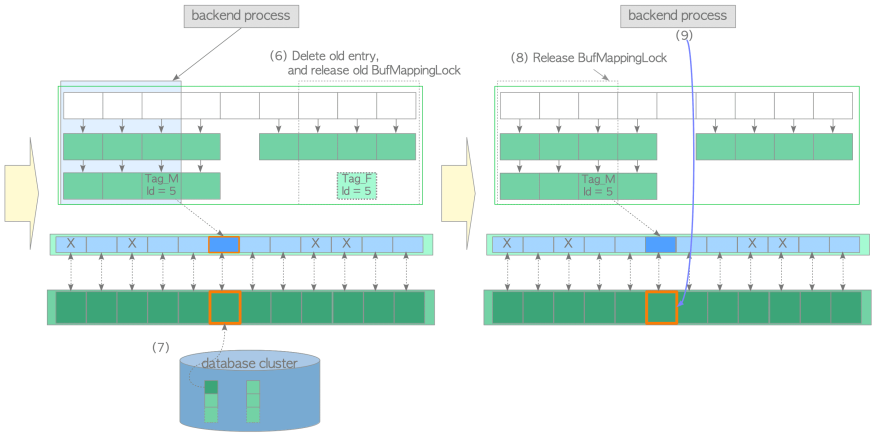

Figure 8.11. Loading a page from storage to a victim buffer pool slot (continued from Figure 8.10).

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

Pseudocode: clock-sweep

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

- (1) Obtain the candidate buffer descriptor pointed 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 Figure 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.

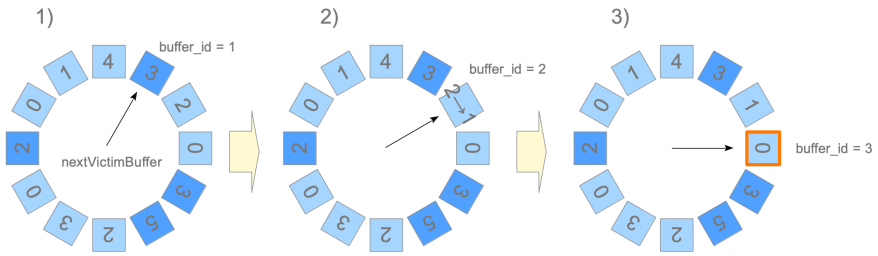

Figure 8.12. Clock Sweep.

- 1) The nextVictimBuffer points to the first descriptor (buffer_id 1). However, this descriptor is skipped because it is pinned.
- 2) The nextVictimBuffer points to the second descriptor (buffer_id 2). This descriptor is unpinned but its usage_count is 2. Thus, the usage_count is decreased by 1, and the nextVictimBuffer advances to the third candidate.
- 3) The nextVictimBuffer points to the third descriptor (buffer_id 3). This descriptor is unpinned and its usage_count is 0. Thus, this is the victim in this round.

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

8.5. RING BUFFER AND LOCAL BUFFER

This section introduces two types of temporary buffers:

- Ring Buffer: Allocated in shared memory
- Local Buffer: Allocated in each backend's memory

8.5.1. 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, temporary buffer area. It is allocated in shared memory when any of the following conditions is met:

1. Bulk-reading:
   When scanning a relation whose size exceeds one-quarter of the buffer pool size ($\text{shared\_buffers}/4$). In this case, the ring buffer size is 256 KB.

2. Bulk-writing:
   When executing the following SQL commands, 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 process performs vacuuming. In this case, the ring buffer size is 256 KB.

The ring buffer is released immediately after use.

The benefit of the ring buffer is clear: If a backend process reads a large table without a ring buffer, all pages stored in the buffer pool may be evicted, reducing the cache hit ratio. The ring buffer prevents this by providing a temporary buffer area for large tables.

Why the default ring buffer size for bulk-reading and vacuum processing is 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)*

When the same relation is accessed concurrently by two or more backends, the ring buffer is shared among them.

Figure 8.13 illustrates how two backends access the ring buffer using a sequential scan:

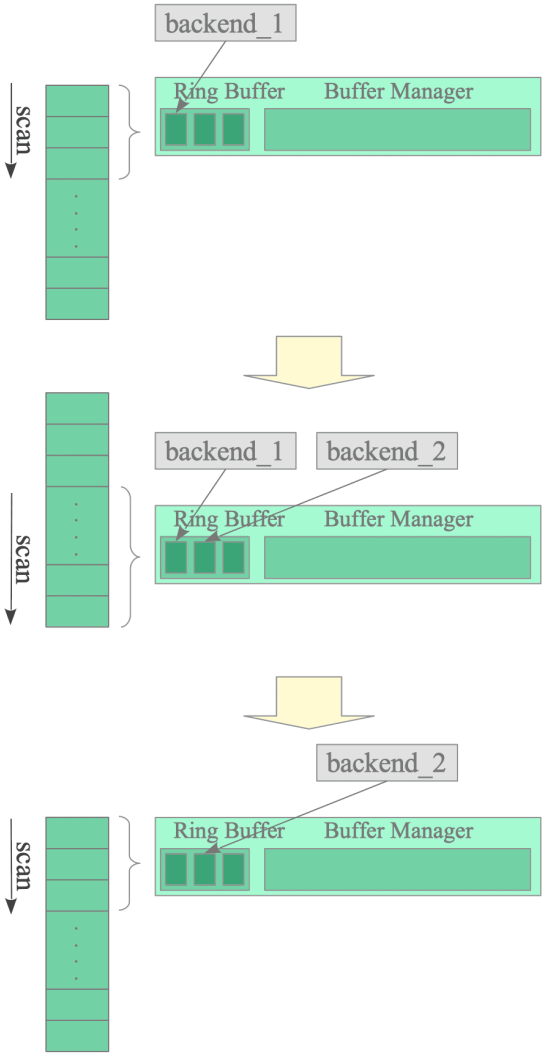

Figure 8.13. Sharing the Ring Buffer with Two Backends.

1. Backend_1 accesses a table via the ring buffer, which is created when Backend_1 accesses the table.
2. Backend_1 and Backend_2 access the table loaded into the ring buffer. Backend_2 starts its sequential scan from the middle of the table.
3. After Backend_1 completes its scan, Backend_2 continues scanning the rest of the table, starting from the beginning and proceeding to the middle.

8.5.2. Temporary Tables and Local Buffer Management

When a backend creates a temporary table, the buffer manager allocates a memory area for the backend and creates a local buffer.

While the numbering of bucket slots for the shared buffer is positive, the numbering of the local bucket slots for the local buffer is negative, i.e., -1, -2, and so on. This allows the backend to seamlessly access both regular tables and temporary tables.

Managing local buffers does not require locks, as these buffers are accessed exclusively by the backend that creates them. Additionally, local buffers do not need to be WAL-logged or checkpointed.

8.6. FLUSHING DIRTY PAGES

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

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

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

Why the checkpointer was separated from the background writer?

In versions 9.1 or earlier, background writer had regularly done the checkpoint processing.

In version 9.2, the checkpointer process has been separated from the background writer process. Since the reason is described in the proposal whose title is "Separating bgwriter and checkpointer", the sentences from it are shown in the following.

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

9. WRITE AHEAD LOGGING (WAL)

Transaction 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 Chapters 10 and 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.

Chapter Contents

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 Figure 9.1.

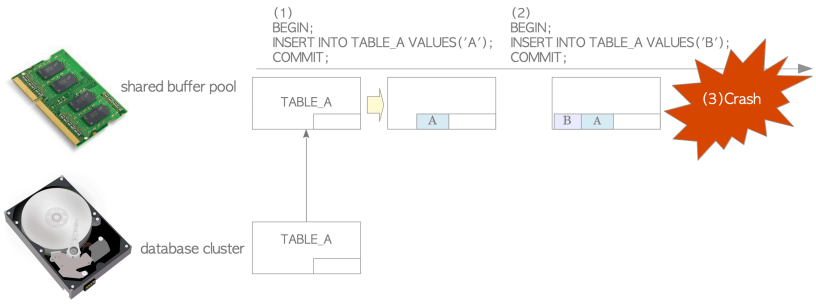

Figure 9.1. Insertion operations without WAL.

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.

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.

Info

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 Figure 9.2 and the following description.

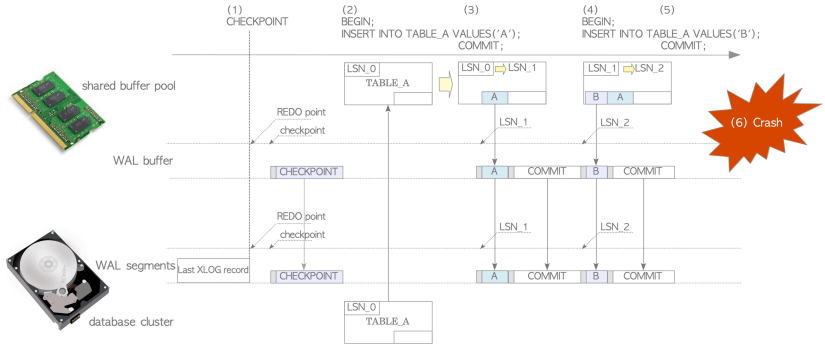

Figure 9.2. Insertion operations with WAL.

Notation

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

1. A checkpointer, a background process, periodically performs checkpointing. Whenever the checkpointer starts, it writes a XLOG record called **checkpoint record** to the current WAL segment. This record contains the location of the latest *REDO point*.
2. 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 Figure 9.3. PostgreSQL will sequentially read and replay XLOG records within the appropriate WAL segment files from the REDO point.

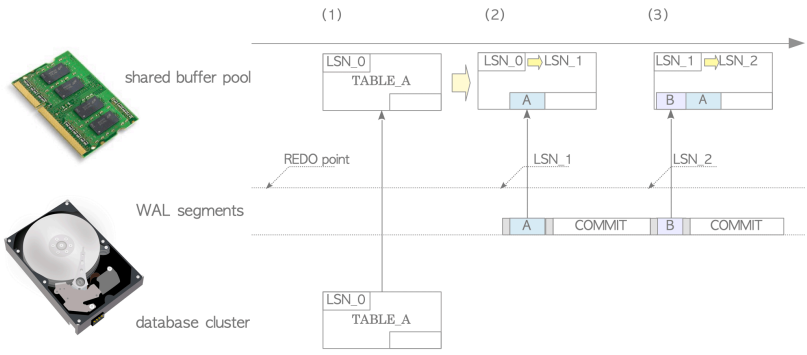

Figure 9.3. Database recovery using WAL.

1. PostgreSQL reads the XLOG record of the first INSERT statement from the appropriate WAL segment file, 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**.

> **Note**
>
> 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 Figure 9.4 and the following description.

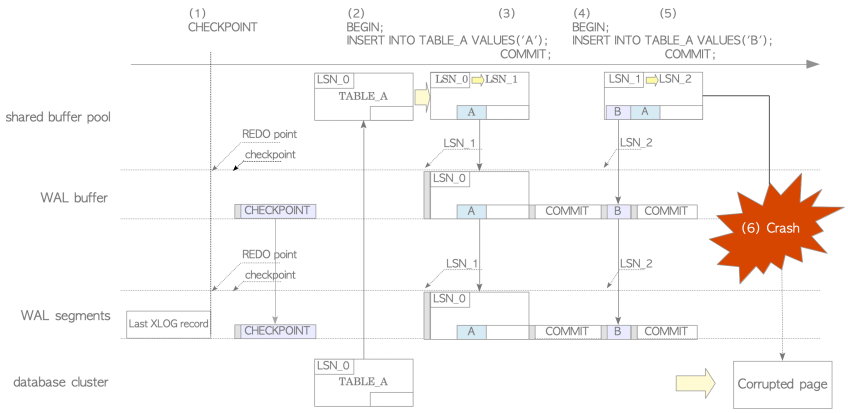

Figure 9.4. Full page writes.

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 Figure 9.5 and the following description.

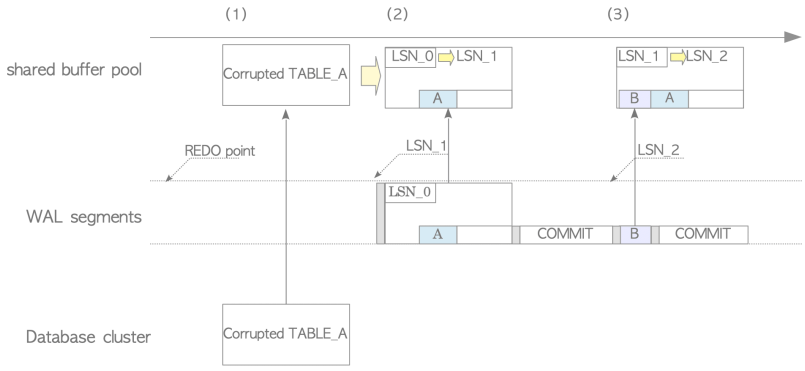

Figure 9.5. Database recovery with backup block.

1. PostgreSQL reads the XLOG record of the first INSERT statement and loads the corrupted TABLE_A 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.

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.

Can Synchronous Replication Eliminate the Need for Backups?

No. Synchronous replication does not eliminate the need for backups.

Operating a DBMS requires addressing not only hardware or software failures but also data loss and errors caused by human mistakes or software bugs.

For instance, if critical data is accidentally deleted, the deletion is immediately reflected on the replicated standby server. Similarly, if erroneous data is written due to a mistake, the error propagates instantly across all replication servers.

In such cases, replication cannot restore the lost or corrupted data. The only way to recover from these critical failures is by using backup data (+ archive logs).

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 Figure 9.6.

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.

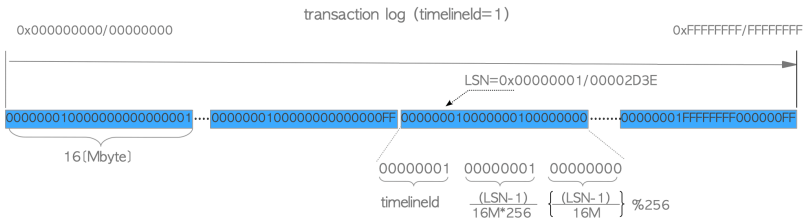

Figure 9.6. Transaction log and WAL segment files

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

$$\text{WAL segment file name} = \text{timelineId} + (\text{uint32})\frac{\text{LSN}-1}{16\text{M}*256} + (\text{uint32})\left(\frac{\text{LSN}-1}{16\text{M}}\right)$$

timelineId

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

The first WAL segment file is 000000010000000000000001. If the first one has been filled up with the writing of XLOG records, the second one 000000010000000000000002 would be provided. Files are used in ascending order in succession. After 00000001000000000000000FF 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 00000001000000010000000FF 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* (versions 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 Figure 9.7.

XLogLongPageHeaderData

```
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_blcksz;    /* just as a cross-check */
} XLogLongPageHeaderData;
```

XLogPageHeaderData

```
/*
 * 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;
```

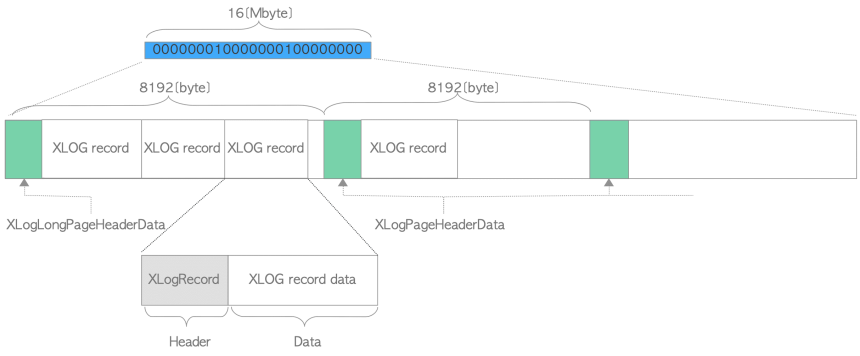

Figure 9.7. Internal layout of a WAL segment file.

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;
```

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 `XLogRecord` structure to refine the XLOG record format, which reduced the size by a few bytes.

XLogRecord

```
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.

Info

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).

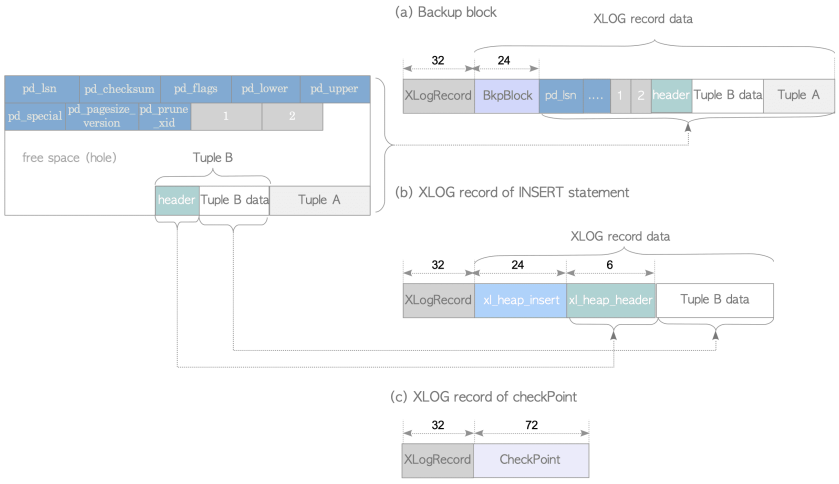

Figure 9.8. Examples of XLOG records (versions 9.4 or earlier).

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 Figure 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 the 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.

BkpBlock

```
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 Figure 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.

xl_heap_insert

```
typedef struct BlockIdData
{
   uint16          bi_hi;
   uint16          bi_lo;
} BlockIdData;

typedef uint16 OffsetNumber;

typedef struct ItemPointerData
{
   BlockIdData     ip_blkid;
   OffsetNumber    ip_posid;
}

typedef struct RelFileNode
{
   Oid             spcNode;            /* tablespace */
   Oid             dbNode;             /* database */
   Oid             relNode;            /* relation */
} RelFileNode;

typedef struct xl_heaptid
{
   RelFileNode     node;
   ItemPointerData tid;                /* changed tuple id */
} xl_heaptid;

typedef struct xl_heap_insert
{
   xl_heaptid      target;             /* inserted tuple id */
   bool            all_visible_cleared; /* PD_ALL_VISIBLE was cleared */
} xl_heap_insert;
```

Info

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 Figure 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).

Info

The xl_heap_header structure (versions 9.4 or earlier) is defined in src/include/access/heapam_xlog.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 Figure 9.9.

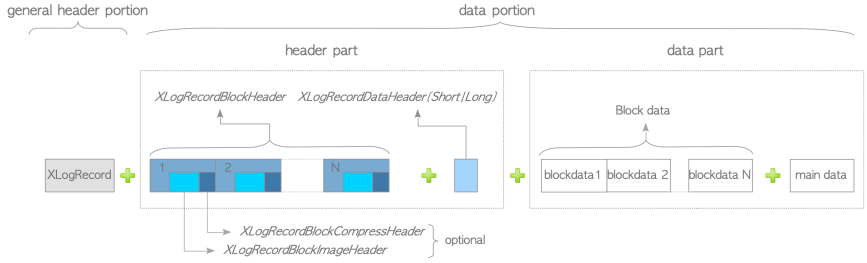

Figure 9.9. Common XLOG record format.

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.

XLogRecordBlockHeader

```
/*
 * 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 */
```

XLogRecordBlockImageHeader

```
/*
 * 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 - <length of "hole" bytes>).
 *
 * 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 BKPIMAGE_HAS_HOLE and BKPIMAGE_COMPRESSED(), an
     * XLogRecordBlockCompressHeader struct follows.
     */
} XLogRecordBlockImageHeader;

/* Information stored in bimg_info */
#define BKPIMAGE_HAS_HOLE       0x01    /* page image has "hole" */
#define BKPIMAGE_APPLY          0x02    /* page image should be restored
                                         * during replay */
/* compression methods supported */
#define BKPIMAGE_COMPRESS_PGLZ  0x04
#define BKPIMAGE_COMPRESS_LZ4   0x08
#define BKPIMAGE_COMPRESS_ZSTD  0x10

#define BKPIMAGE_COMPRESSED(info) \
    ((info & (BKPIMAGE_COMPRESS_PGLZ | BKPIMAGE_COMPRESS_LZ4 | \
              BKPIMAGE_COMPRESS_ZSTD)) != 0)
```

XLogRecordBlockCompressHeader

```
/*
 * Extra header information used when page image has "hole" and
 * is compressed.
 */
typedef struct XLogRecordBlockCompressHeader
```

```
{
    uint16      hole_length;    /* number of bytes in "hole" */
} XLogRecordBlockCompressHeader;
```

XLogRecordDataHeader

```
/*
 * 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.

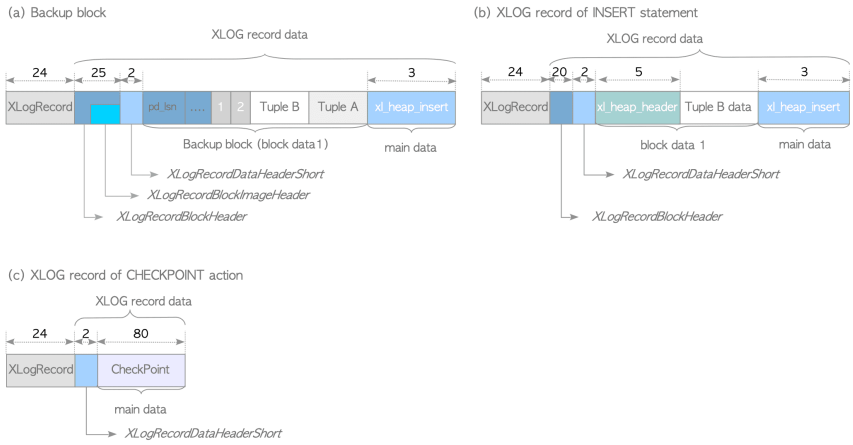

Figure 9.10. Examples of XLOG records (versions 9.5 or later).

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 Figure 9.10(a). It is composed of four data structures and one data object:

1. the XLogRecord structure (header-portion).
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.)

> **Info**
>
> 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, we will describe the non-backup block record created by the INSERT statement (see Figure 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 relfilenode, 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.

xl_heap_insert

```
typedef struct xl_heap_insert
{
        OffsetNumber    offnum;           /* inserted tuple's offset */
        uint8           flags;

        /* xl_heap_header & TUPLE DATA in backup block 0 */
} xl_heap_insert;
```

As the final example, a checkpoint record is shown in the Figure 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).

> **Info**
>
> 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.

The new XLOG format, while more complex for human interpretation, is optimized for parser efficiency. Additionally, many XLOG record types are now smaller in size.

The sizes of the main structures are shown in Figures 9.8 and 9.10, allowing for the calculation and comparison of record sizes[1].

---

1. While the size of the new checkpoint is larger than the previous one, it includes 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. This section will provide a comprehensive overview of the process.

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.
                                           */
(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.
                                           */
```

The following paragraphs explain each line of the pseudocode to illustrate XLOG record writing. For a visual representation, refer to Figures 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 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.

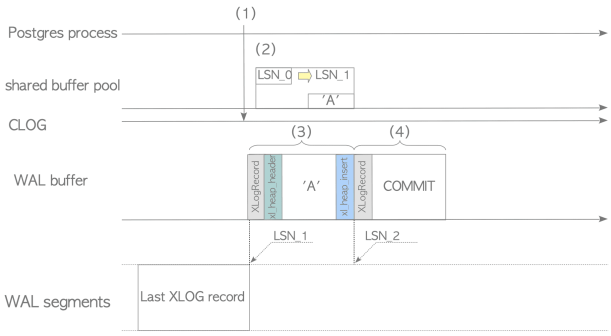

Figure 9.11. Write-sequence of XLOG records.

The format of these XLOG records is version 9.4.

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

6. The function TransactionIdCommitTree() changes the state of this transaction from 'IN_PROGRESS' to 'COMMITTED' on the CLOG.

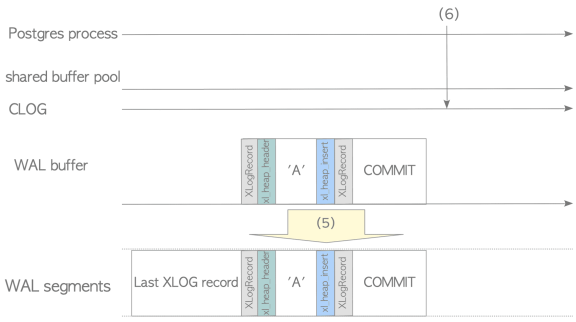

Figure 9.12. Write-sequence of XLOG records. (continued from Figure 9.11)

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.

Direct I/O

PostgreSQL versions 15 or 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.

9.5.1. Remark on Writing XLOG records

It is well understood that DML (Data Manipulation Language) operations generate XLOG records. However, non-DML operations can also create XLOG records. For instance:

- A commit action writes an XLOG record containing the ID of the committed transaction.
- A checkpoint action writes an XLOG record containing general information about the checkpoint.

In special cases, even SELECT statements can generate XLOG records, despite typically not doing so. For example:

- A SELECT FOR UPDATE statement generates XLOG records for all target tuple locks (**ROW SHARE LOCK**) [1].
- During Heap-Only Tuple (HOT) operations, where unnecessary tuples are deleted and necessary tuples are defragmented within pages, XLOG records for the modified pages are written to the WAL buffer.

Additionally, if the wal_level parameter is set to *replica* or higher, **ACCESS EXCLUSIVE LOCKS** are also recorded as XLOG records. This occurs not only when an ACCESS EXCLUSIVE LOCK is explicitly acquired using the LOCK command but also when commands like DROP TABLE and TRUNCATE (which acquire EXCLUSIVE LOCKS) are executed, as described in the official document.

---

1. Occasionally, users report unexpected increases in WAL segment consumption despite executing only SELECT commands. In such cases, check whether SELECT FOR UPDATE commands are being run. ↩

9.6. WAL RELATED PROCESSES

9.6.1. 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.

9.6.2. WAL Summarizer Process

Introduced in version 17 to support the incremental backups (described in Section 10.5). the WAL Summarizer process tracks changes to all database blocks (including relations and visibility maps) and writes these modifications to WAL summary files located in the `pg_wal/summaries/` directory.

Note that the free-space map fork is not tracked because it is not properly WAL-logged.

9.6.2.1. Outline of how WAL Summarizer process works

The WAL Summarizer process operates as follows:

1. During each checkpoint, it reads WAL segment files from the previous REDO point to the current REDO point.
2. It tracks changes to all blocks of all relations (including visibility maps) using the WAL segment files.
3. It writes the summary to WAL summary files located in the pg_wal/summaries/ directory.

In the context of the WAL Summarizer process, “previous REDO point” and “current REDO point” are referred to as “start_lsn” and “end_lsn,” respectively.

The summary file name format is as follows:

```
{Timeline}{start_lsn}{end_lsn}.summary
```

Here’s an example of summary files:

```
$ ls -1 data/pg_wal/summaries/
000000010000000000100002800000000010B1D30.summary
000000010000000000010B1D300000000001473DE0.summary
000000010000000001473DE00000000001473EE0.summary
000000010000000001473EE0000000000147A8A8.summary
000000010000000000147A8A8000000000147A9A8.summary

... snip ...
```

The pg_available_wal_summaries() function can show the wal summaries:

```
testdb=# SELECT tli, start_lsn, end_lsn FROM pg_available_wal_summaries() ORDER BY start_lsn;
 tli |  start_lsn  |   end_lsn
-----+-------------+-------------
   1 | 0/1000028   | 0/10B1D30
   1 | 0/10B1D30   | 0/1473DE0
   1 | 0/1473DE0   | 0/1473EE0
   1 | 0/1473EE0   | 0/147A8A8
   1 | 0/147A8A8   | 0/147A9A8

... snip ...
```

Summary files are automatically removed after the configured wal_summary_keep_time (default 10 days) has passed since their creation.

9.6.2.2. Contents of a Summary File

To illustrate the contents of a summary file, we will create four tables, t1, t2, t3, and t4, each containing of four blocks:

```
testdb=# CREATE TABLE t1 (id int);
CREATE TABLE
testdb=# INSERT INTO t1 SELECT  GENERATE_SERIES(1, 800);
INSERT 0 800
testdb=# SELECT * FROM pg_freespace('t1');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
```

```
     2 |     0
     3 |     0
(4 rows)

testdb=# CREATE TABLE t2 (id int);
CREATE TABLE
testdb=# INSERT INTO t2 SELECT  GENERATE_SERIES(1, 800);
INSERT 0 800
testdb=# CREATE TABLE t3 (id int);
CREATE TABLE
testdb=# INSERT INTO t3 SELECT  GENERATE_SERIES(1, 800);
INSERT 0 800
testdb=# CREATE TABLE t4 (id int);
CREATE TABLE
testdb=# INSERT INTO t4 SELECT  GENERATE_SERIES(1, 800);
INSERT 0 800
testdb=# CHECKPOINT;
CHECKPOINT
```

After a CHECKPOINT, we will perform the following operations:

1. Update two rows in t1.
2. Insert 150 rows into t2.
3. Delete 300 rows from t3.
4. Truncate all rows from t4.
5. Create a new table t5 and insert 800 rows into it.

```
testdb=# -- [1] Update two rows to modify the blocks of t1
testdb=# UPDATE t1 SET id = id + 1000 WHERE id = 1 OR id = 200;
UPDATE 2
testdb=# -- [2] Insert 150 rows to modify the last block and add a new block
testdb=# INSERT INTO t2 SELECT GENERATE_SERIES(1, 150);
INSERT 0 150
testdb=# -- [3] Delete 500 rows to remove blocks
testdb=# DELETE FROM t3 WHERE id > 300;
DELETE 500
testdb=# -- [4] Truncate all blocks
testdb=# TRUNCATE t4;
TRUNCATE TABLE
testdb=# -- [5] Create new table
testdb=# CREATE TABLE t5 (id int);
CREATE TABLE
testdb=# INSERT INTO t5 SELECT  GENERATE_SERIES(1, 800);
INSERT 0 800
testdb=# CHECKPOINT;
CHECKPOINT
```

Following these operations, we will examine the summary of changes to all relation blocks generated by the WALsummarizer process, using the pg_wal_summary_contents() function.

The `pg_wal_summary_contents(timeline, start_lsn, end_lsn)` function shows all changed blocks from start_lsn to end_lsn. The output contains: filenode (OID), block number, fork number, is_limit_block (explained later), etc.

[1] Modified blocks

Table t1 has been updated for two rows.

```
testdb=# UPDATE t1 SET id = id + 1000 WHERE id = 1 OR id = 200;
UPDATE 2
```

Using the pg_wal_summary_contents() function, we can retrieve the summary data for table t1 as follows:

```
testdb=# SELECT p.relname, s.relforknumber, s.relblocknumber, s.is_limit_block
testdb-#    FROM pg_wal_summary_contents(1, '0/1F4225F8', '0/1F476450') AS s, pg_class AS p
testdb-#    WHERE s.relfilenode = p.oid AND p.relname = 't1';
 relname | relforknumber | relblocknumber | is_limit_block
---------+---------------+----------------+----------------
 t1      |             0 |              0 | f
 t1      |             0 |              3 | f
(2 rows)
```

The output shows that the 0th and 3rd blocks of table t1 have been modified due to the UPDATE command.

Figure 9.13 illustrates the modification of table t1, which reflects the summary data for t1.

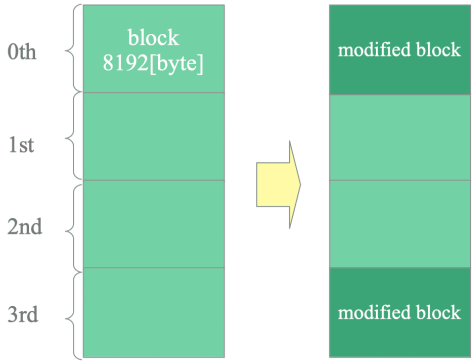

Figure 9.13. The modification of table t1.

[2] Added blocks

Table t2 has 150 new rows added.

As a result, we can confirm that the 3rd block was changed and the new 4th block was added.

```
testdb=# SELECT p.relname, s.relforknumber, s.relblocknumber, s.is_limit_block
testdb-#    FROM pg_wal_summary_contents(1, '0/1F4225F8', '0/1F476450') AS s, pg_class AS p
testdb-#    WHERE s.relfilenode = p.oid AND p.relname = 't2';
 relname | relforknumber | relblocknumber | is_limit_block
---------+---------------+----------------+----------------
 t2      |             0 |              3 | f
 t2      |             0 |              4 | f
(2 rows)

testdb=# select * from pg_freespace('t2');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |     0
     4 |     0
(5 rows)
```

Figure 9.14 illustrates the modification of table t2, which reflects the summary data for t2.

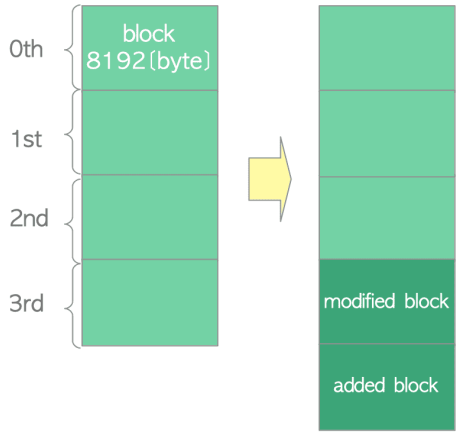

Figure 9.14. The modification of table t2.

[3] Removed blocks

Table t3 has deleted 500 rows.

```
testdb=# SELECT p.relname, s.relforknumber, s.relblocknumber, s.is_limit_block
testdb-#    FROM pg_wal_summary_contents(1, '0/1F4225F8', '0/1F476450') AS s, pg_class AS p
testdb-#    WHERE s.relfilenode = p.oid AND p.relname = 't3';
 relname | relforknumber | relblocknumber | is_limit_block
---------+---------------+----------------+----------------
 t3      |             0 |              2 | t
 t3      |             0 |              1 | f
 t3      |             0 |              0 | f
```

```
 t3      |              2 |              2 | t
 t3      |              2 |              0 | f
(5 rows)

testdb=# select * from pg_freespace('t3');
 blkno | avail
-------+-------
     0 |     0
     1 |  5472
(2 rows)
```

According to the output, the 2nd and 3rd blocks were removed, and the remaining blocks, i.e., 0th and 1st blocks, were modified. Moreover, the 2nd block of t3's visibility map was removed, and the 0th block was modified.

In this way, if blocks after a certain block number were deleted, the block on the boundary of the deleted blocks is also recorded, and **is_limit_block** is set to **true**.

In this case, the 2nd block of table t3 and the 2nd block of table t3's visibility space map are both set to **is_limit_block = true**, because both the 2nd block and the subsequent blocks were deleted.

Figure 9.15 illustrates the modification of table t3, which reflects the summary data for t3. As shown in Figure 9.15, the limit block acts as a virtual termination block.

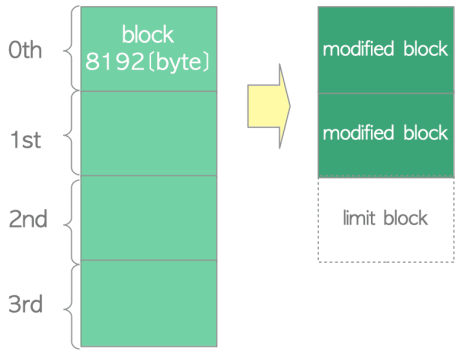

Figure 9.15. The modification of table t3.

[4] Truncated all blocks

Since the truncated table t4 and visibility-map files were removed, the block number for all related blocks was set to 0, and **is_limit_block** was set to **true**.

```
testdb=# SELECT p.relname, s.relforknumber, s.relblocknumber, s.is_limit_block
testdb-#    FROM pg_wal_summary_contents(1, '0/1F4225F8', '0/1F476450') AS s, pg_class AS p
testdb-#    WHERE s.relfilenode = p.oid AND p.relname = 't4';
 relname | relforknumber | relblocknumber | is_limit_block
---------+---------------+----------------+----------------
 t4      |             0 |              0 | t
 t4      |             2 |              0 | t
 t4      |             3 |              0 | t  <= fork_num 3 is a special number,
                                              so the explanation is omitted here.
(3 rows)

testdb=# select * from pg_freespace('t4');
 blkno | avail
-------+-------
(0 rows)
```

Figure 9.16 illustrates the modification of table t4, which reflects the summary data for t4.

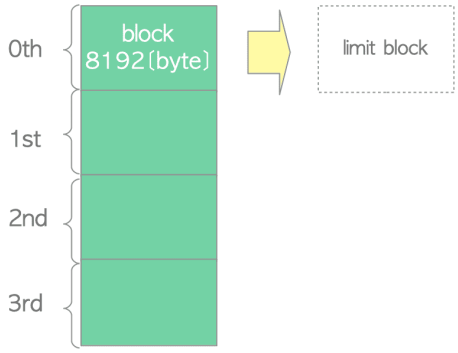

Figure 9.16. The modification of table t4.

This same result returns when DROP TABLE is executed.

[5] Created new table

Not only when data blocks are removed, but also when a new table is created, the block number of the created relation is set to 0, and **is_limit_block** is set to **true**.

In this case, we created table t5 and inserted rows into it. Therefore, the summary file contains two rows that are both for the 0th block of table t5, but **is_limit_block** is **true** and **false**, respectively.

```
testdb=# SELECT p.relname, s.relforknumber, s.relblocknumber, s.is_limit_block
testdb-#    FROM pg_wal_summary_contents(1, '0/1F4225F8', '0/1F476450') AS s, pg_class AS p
testdb-#    WHERE s.relfilenode = p.oid AND p.relname = 't5';
 relname | relforknumber | relblocknumber | is_limit_block
---------+---------------+----------------+----------------
 t5      |             0 |              0 | t
 t5      |             0 |              0 | f
 t5      |             0 |              1 | f
 t5      |             0 |              2 | f
 t5      |             0 |              3 | f
 t5      |             2 |              0 | f
(6 rows)
```

Figure 9.17 illustrates the modification of table t5, which reflects the summary data for t5.

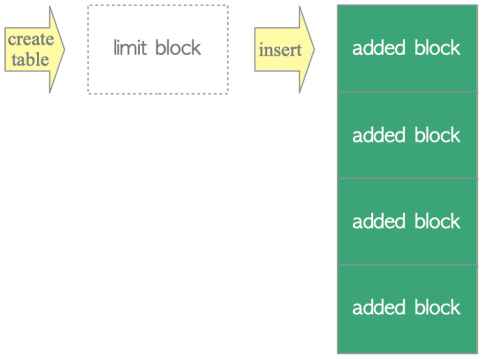

Figure 9.17. The modification of table t5.

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).
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.

Info

In versions 9.1 or earlier, as mentioned in Section 8.6, the background writer process did both checkpinting and dirty-page writing.

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

9.7.1. Outline of the Checkpoint Processing

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 Figure 9.18 for an overview.

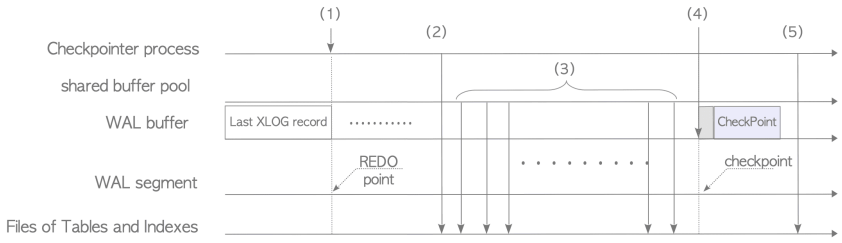

Figure 9.18. Internal processing of PostgreSQL Checkpoint.

1. When a checkpoint process starts, the REDO point is stored in memory. The REDO point is the location of the XLOG record written at the moment this checkpoint process began. It is the starting point for database recovery.
2. All data in shared memory (e.g., the contents of the clog, etc..) is flushed to the storage.
3. All dirty pages in the shared buffer pool are gradually written and flushed to the storage.
4. 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*.
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.

› struct CheckPoint

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 ℹ 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.

```
$ 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:             Mon Jan 1 15:16:38 2024
Latest checkpoint location:           0/16AF0090
Latest checkpoint's REDO location:    0/16AF0090
Latest checkpoint's REDO WAL file:    000000010000000000000016

... snip ...
```

ℹ 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 discussed database recovery several times. This section will cover two additional aspects of recovery.

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 Figure 9.19 and the following description.

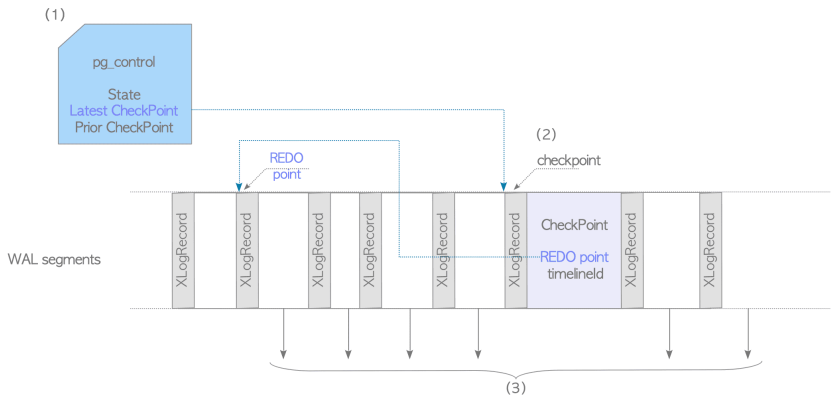

Figure 9.19. Details of the recovery process.

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 Figures 9.20 and 9.21. (Note that the WAL buffer is omitted to simplify the description.)

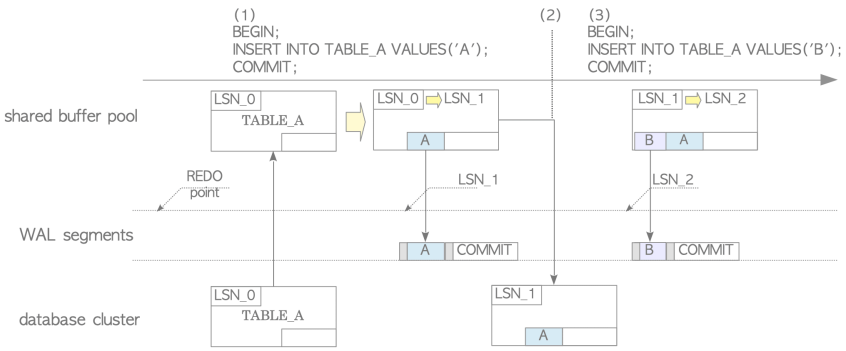

Figure 9.20. Insertion operations during the background writer working.

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.

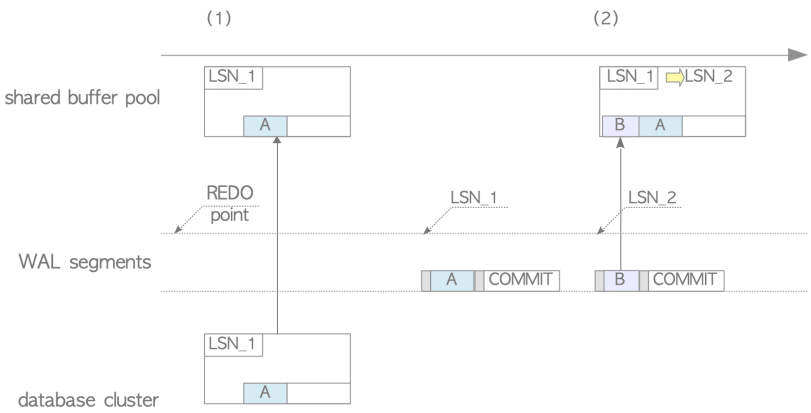

Figure 9.21. Database recovery.

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).

In this example, if the replay order of non-backup blocks is incorrect or non-backup blocks are replayed more than once, the database cluster becomes inconsistent. This highlights that the redo (replay) operation for non-backup blocks is **not idempotent**. To ensure the correct replay order, non-backup block records should only be replayed when their LSN is greater than the corresponding page's pd_lsn.

In contrast, the redo operation for backup blocks is **idempotent**, meaning these blocks can be replayed multiple 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 Figure 9.22. 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.

Info

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

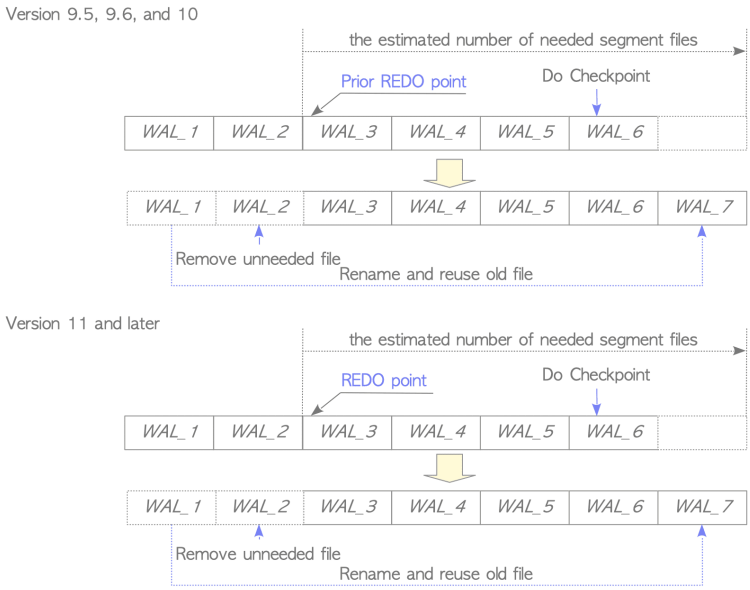

Figure 9.22. Recycling and removing WAL segment files at a checkpoint.

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 Figure 9.23, if WAL_7 has been filled up, WAL_8 will be newly created.

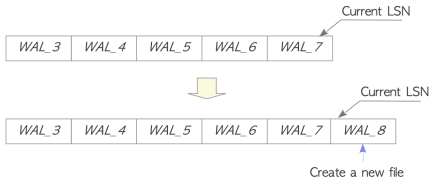

Figure 9.23. Creating WAL segment file.

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 9.24 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.

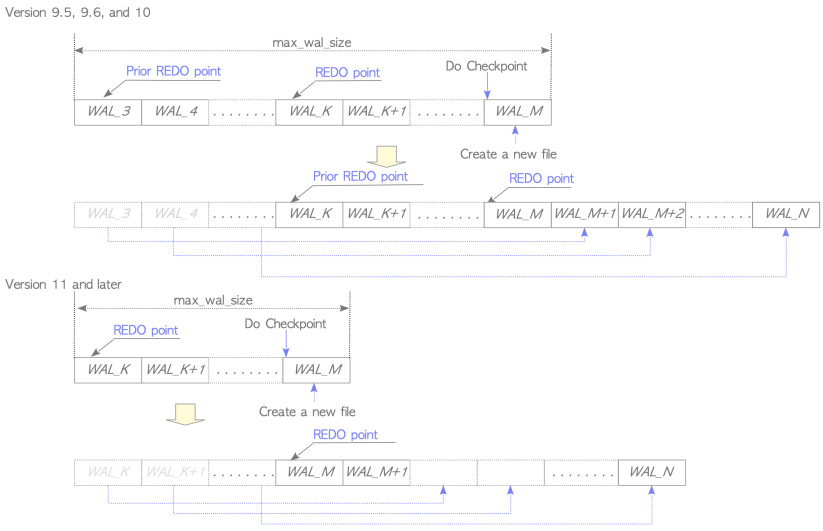

Figure 9.24. Checkpointing and recycling WAL segment files.

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.

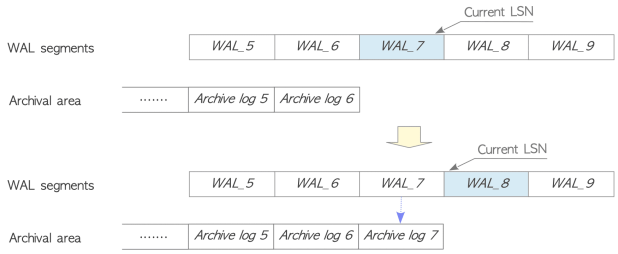

Figure 9.25. Continuous archiving.

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 enables the use of scp or other file backup tools to transfer archive logs to remote hosts, instead of relying on simple copy commands.

archive_library

PostgreSQL versions 14 and earlier relied on shell commands for continuous archiving.

However, version 15 introduced a loadable library feature, enabling continuous archiving through library-based mechanisms.

For further details, consult the documentation on archive_library and basic_archive.

Note

PostgreSQL does not automatically clean up created archive logs, so proper log management is essential when using this feature. Without intervention, the number of archive logs will continuously grow.

The pg_archivecleanup utility is one of the useful tools for managing archive log files.

Additionally, the *find* command can be used to delete old archive logs.

For instance, to remove logs older than three days:

```
$ find /home/postgres/archives -mtime +3d -exec rm  -f {} \;
```

10. ONLINE BACKUP AND POINT-IN-TIME RECOVERY (PITR)

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 version 17, PostgreSQL has finally supported incremental backup feature.

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?
- How does Incremental backup in PostgreSQL work?

Chapter Contents

Historical Info

In versions 7.4 or earlier, PostgreSQL had supported only logical backups (logical full and partial backups, and data exports).

10.1. BASE BACKUP

Before the introduction of the pg_basebackup utility in version 9.1, online (full) backups were performed using the pg_backup_start and pg_backup_stop commands (versions 8.0 to 9.0). While these commands are no longer commonly used, understanding them is essential for comprehending PostgreSQL's backup and Point-in-Time Recovery (PITR) mechanisms. We will explore them in more detail in the following subsections and then discuss how pg_basebackup works.

Figure 10.1 shows the standard procedure for taking a base backup using these 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).

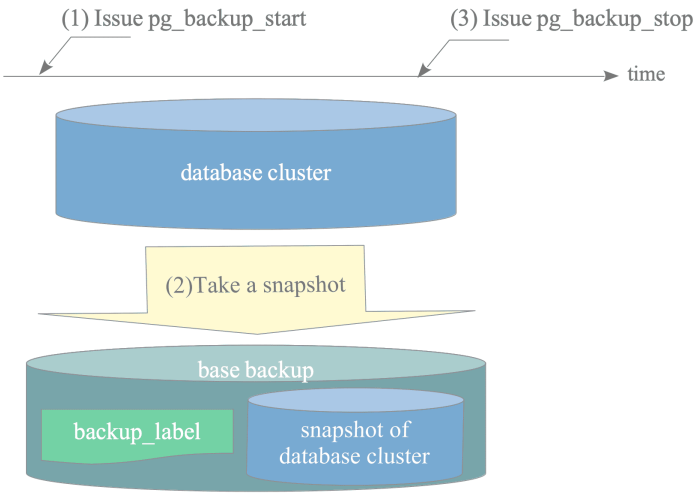

Figure 10.1. Making a base backup.

This procedure does not require table locks, allowing all users to continue issuing queries without being affected by the backup operation. This is a significant advantage over other major open-source RDBMSs.

The pg_basebackup utility internally invokes the commands described above and therefore inherits their advantages.

Info

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, internally invokes do_pg_backup_start() function, 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:

```
$ cat /usr/local/pgsql/data/backup_label
START WAL LOCATION: 0/1B000028 (file 000000010000000000000001B)
CHECKPOINT LOCATION: 0/1B000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2024-1-1 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, internally invokes do_pg_backup_stop() function, 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.

Info

The naming method for backup history file is shown below.

```
{WAL segment}.{offset value at the time the base backup was started}.backup
```

10.1.3. pg_basebackup

pg_basebackup is a utility for taking online backups.

In version 16 or earlier, pg_basebackup supported full backups of the entire database cluster. In version 17, incremental backups were added. This section explains how to take a full backup. Incremental backups will be discussed in Section 10.5.

To perform a backup remotely, pg_basebackup utilizes the walsender process, a component of streaming replication, which will be explained in Chapter 11.

For instance, to take a full backup from host 192.168.1.10 to the local server at '/usr/local/pgsql/backup/full', we can execute the following command:

```
$ pg_basebackup -h 192.168.1.10 -p 5432 -D /usr/local/pgsql/backup/full -X stream -P -v
```

Figure 10.2 shows the sequence of how the pg_basebackup takes a full backup:

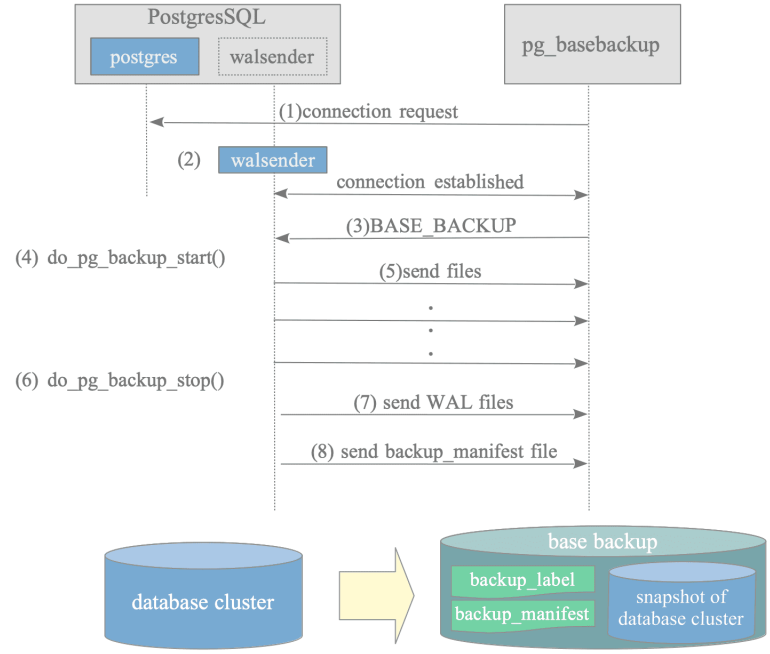

Figure 10.2. The sequence of how the pg_basebackup takes a full backup.

1. Connection request:
   pg_basebackup requests the PostgreSQL server to connect a walsender process.
2. Create walsender process:
   The PostgreSQL server creates a walsender process and establishes a connection between the walsender and pg_basebackup.
3. Base backup request:
   pg_basebackup requests a base backup.
4. Do do_pg_backup_start():
   The walsender process executes the do_pg_backup_start() function.
5. Send all files:
   The walsender process sends all files under the database cluster except WAL segment files in the pg_wal directory.
6. Do do_pg_backup_stop():
   The walsender process executes the do_pg_backup_stop() function.
7. Send WAL files:
   The walsender process sends all WAL segment files in the pg_wal directory if the '–wal-method' option is set to a value other than 'none'.
8. Send backup_manifest file:
   The walsender process creates and sends the backup_manifest file, which is explained in the following subsection.

The reason why WAL segment files are not sent in step 5 is to ensure that the last WAL segment files will be sent to pg_basebackup. In the step 6, the current WAL segment is switched by executing the do_pg_backup_stop() function, which means that all WAL segment files generated during the backup procedure are certainly stored in the pg_wal directory.

```
$ ls /usr/local/pgsql/backup/full
PG_VERSION        global        pg_ident.conf  pg_serial     pg_tblspc             postgresql.conf
backup_label      log           pg_logical     pg_snapshots  pg_twophase
backup_manifest   pg_commit_ts  pg_multixact   pg_stat       pg_wal
base              pg_dynshmem   pg_notify      pg_stat_tmp   pg_xact
current_logfiles  pg_hba.conf   pg_replslot    pg_subtrans   postgresql.auto.conf
```

Why does pg_basebackup use walsender?

pg_basebackup does not have a direct relationship with replication. When the pg_basebackup utility was developed (and even now), the only processes that could be started as a PostgreSQL server process from an external program were 'postgres' and 'walsender'. Therefore, the walsender protocol was extended to accommodate pg_basebackup.

10.1.3.1. Backup Manifest Files

A backup manifest file is a JSON file that contains metadata and verification information about the backup.

Key Components are shown as follows:

Key	Values
PostgreSQL-Backup-Manifest-Version	Backup manifest version number.
Files	List of objects that contains all file's path, size, checksum, etc.
WAL-Ranges	Timeline and the LSN range during the backup procedure: * **Start-LSN**: The LSN of the REDO point generated by CHECKPOINT when the do_pg_backup_start function is invoked. * **End-LSN**: The LSN of the WAL log created by the do_pg_backup_stop function.
Manifest-Checksum	The checksum value of this manifest file.

Here's a cited example of a backup manifest file:


```
$ cat /usr/local/pgsql/backup/full/backup_manifest
{ "PostgreSQL-Backup-Manifest-Version": 2,
"System-Identifier": 7426689740139212305,
"Files": [
{ "Path": "backup_label", "Size": 225, "Last-Modified": "2024-10-17 10:41:48 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "1950abcb" },
{ "Path": "postgresql.conf", "Size": 30771, "Last-Modified": "2024-10-17 10:29:00 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "a9c769e0" },
{ "Path": "postgresql.auto.conf", "Size": 88, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "536f950b" },
{ "Path": "pg_ident.conf", "Size": 2640, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "0ce04d87" },
{ "Path": "log/postgresql-2024-10-17_193211.log", "Size": 1359, "Last-Modified": "2024-10-17 10:41:48
GMT", "Checksum-Algorithm": "CRC32C", "Checksum": "62ed7886" },
{ "Path": "log/postgresql-2024-10-17_192939.log", "Size": 5631, "Last-Modified": "2024-10-17 10:32:11
GMT", "Checksum-Algorithm": "CRC32C", "Checksum": "e9891cb3" },
{ "Path": "pg_xact/0000", "Size": 8192, "Last-Modified": "2024-10-17 10:41:48 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "4c2ce5fc" },
{ "Path": "pg_hba.conf", "Size": 5711, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "d62da38c" },
{ "Path": "PG_VERSION", "Size": 3, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "64440205" },
{ "Path": "base/4/113", "Size": 8192, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "d1bc40bb" },
{ "Path": "base/4/1417", "Size": 0, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "00000000" },
{ "Path": "base/4/2610_fsm", "Size": 24576, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "b9b5f34f" },
{ "Path": "base/4/3542", "Size": 16384, "Last-Modified": "2024-10-17 10:29:12 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "e7f849bf" },

... snip ...

{ "Path": "global/pg_control", "Size": 8192, "Last-Modified": "2024-10-17 10:41:48 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "43872087" }
],
"WAL-Ranges": [
{ "Timeline": 1, "Start-LSN": "0/4000028", "End-LSN": "0/4000120" }
],
"Manifest-Checksum": "4c6d8a85379990904f6986f5bfd98db9f4640cfc96f440f8674abe6251cfffb8"}
```

10.2. HOW POINT-IN-TIME RECOVERY WORKS

Figure 10.3 illustrates the basic concept of PITR.

In PITR mode, PostgreSQL replays the WAL data from the archive logs on the base backup, starting at the REDO point created by pf_backup_start and continuing up to the specified recovery point. This recovery point in PostgreSQL is referred to as the *recovery target*.

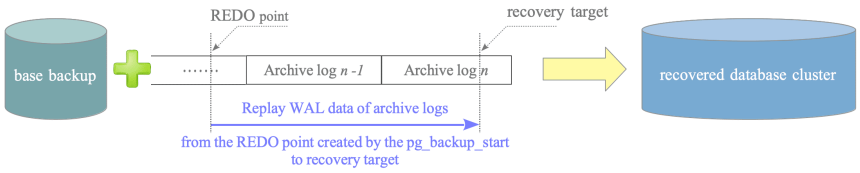

Figure 10.3. Basic concept of PITR.

The process of PITR operates as follows:

Suppose a mistake occurred at 12:05 GMT on 1 January 2024. The database cluster should be removed, and a new one restored using the base backup created before that time.

To begin, configure the command for the restore_command parameter and set the time for the recovery_target_time parameter to the point of the error (in this case, 12:05 GMT) in the postgresql.conf file (or recovery.conf for versions 11 or earlier).

```
# Place archive logs under /mnt/server/archivedir directory.
restore_command = 'cp /mnt/server/archivedir/%f %p'
recovery_target_time = "2024-1-1 12:05 GMT"
```

When PostgreSQL starts up, it enters into PITR mode if there is a 'backup_label' file and a '**recovery.signal**' file (versions 11 or earlier, 'recovery.conf') in the database cluster.

recovery.conf / recovery.signal

The recovery.conf file was removed in version 12, and all recovery-related parameters are now written in postgresql.conf. For detailed information, refer to the official document.

In versions 12 and later, restoring a server from a base backup requires creating an empty file named **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 Section 9.8. The only two differences are:

- 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 archive_command parameter.
- 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 11 or earlier, recovery.conf), 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 '2024-1-1 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 11 or earlier, recovery.conf), 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).

xl_xact_commit

```
typedef struct xl_xact_commit
{
        TimestampTz xact_time;          /* time of commit */
        uint32          xinfo;              /* info flags */
        int             nrels;              /* number of RelFileNodes */
        int             nsubxacts;          /* number of subtransaction XIDs */
        int             nmsgs;              /* number of shared inval msgs */
        Oid             dbId;               /* MyDatabaseId */
        Oid             tsId;               /* MyDatabaseTableSpace */
        /* Array of RelFileNode(s) to drop at commit */
        RelFileNode     xnodes[1];          /* VARIABLE LENGTH ARRAY */
        /* ARRAY OF COMMITTED SUBTRANSACTION XIDs FOLLOWS */
        /* ARRAY OF SHARED INVALIDATION MESSAGES FOLLOWS */
} xl_xact_commit;
```

xl_xact_abort

```
typedef struct xl_xact_abort
{
        TimestampTz     xact_time;          /* time of abort */
        int             nrels;              /* number of RelFileNodes */
        int             nsubxacts;          /* number of subtransaction XIDs */
        /* Array of RelFileNode(s) to drop at abort */
        RelFileNode     xnodes[1];          /* VARIABLE LENGTH ARRAY */
        /* ARRAY OF ABORTED SUBTRANSACTION XIDs FOLLOWS */
} xl_xact_abort;
```

Therefore, if a target time is set to the recovery_target_time parameter, 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.

Info

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.

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. TIMELINEID 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: timelineId and timeline history files.

10.3.1. timelineId

Each timeline is given a corresponding **timelineId**, a 4-byte unsigned integer starting at 1.

An individual timelineId is assigned to each database cluster. The timelineId of the original database cluster created by the initdb utility is 1. Whenever a database cluster recovers, the timelineId is increased by 1. For example, in the example of the previous section, the timelineId of the cluster recovered from the original one is 2.

Figure 10.4 illustrates the PITR process from the viewpoint of the timelineId.

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 (timelineId 1). This situation is represented by the blue arrow line in the figure. Then, a new timelineId 2 is assigned to the recovered database cluster and PostgreSQL runs on the new timeline.

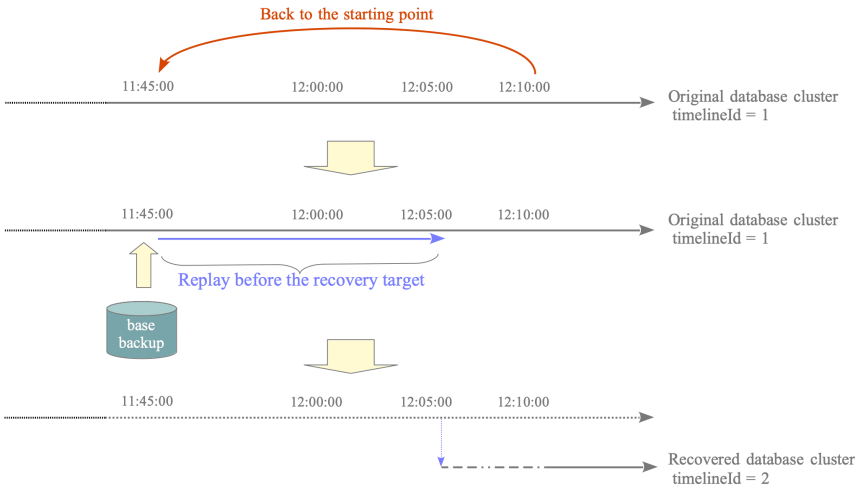

Figure 10.4. Relation of timelineId between an original and a recovered database clusters.

As briefly mentioned in Chapter 9, the first 8 digits of the WAL segment filename are equal to the timelineId of the database cluster that created the segment. When the timelineId 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 '000000020000000000000000A'.

Figure 10.5 shows this situation.

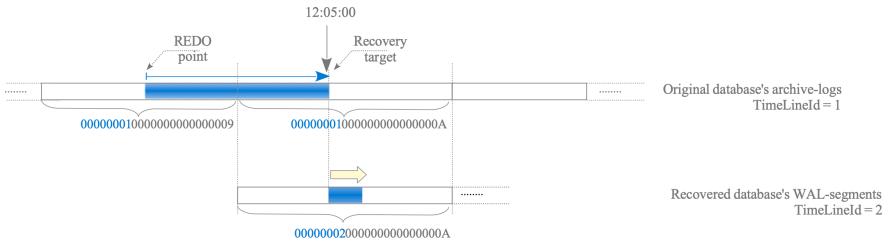

Figure 10.5. Relation of WAL segment files between an original and a recovered database clusters.

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:

```
$ cat /home/postgres/archivelogs/00000002.history
1     0/A000198 before 2024-1-1 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 '2024-1-1 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.

Info

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 is essential for performing second and subsequent Point-in-Time Recovery (PITR) operations. The following example demonstrates how it is utilized during a second recovery attempt.

Consider a scenario where an error occurs at '12:15:00' in the recovered database cluster, which has a timelineId of 2. To recover the database to this point, the postgresql.conf file (or recovery.conf for versions 11 or earlier) should be configured as follows:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
recovery_target_time = "2024-1-1 12:15:00 GMT"
recovery_target_timeline = 2
```

The recovery_target_time parameter specifies the desired recovery time, while the recovery_target_timeline parameter is set to 2 to recover along its timeline.

Restart the PostgreSQL server to enter PITR mode and recover the database at the target time along timeline ID 2 (see Figure 10.6).

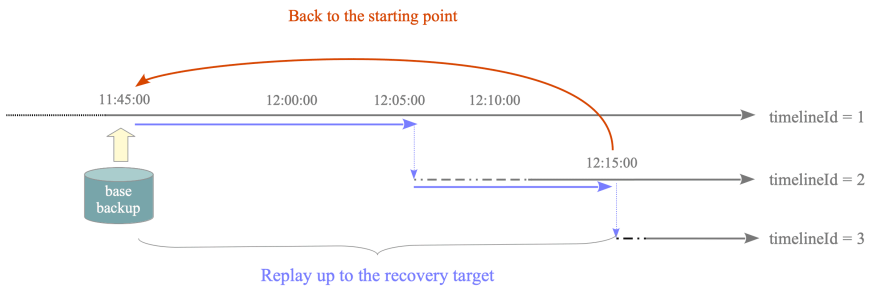

Figure 10.6. Recover the database at 12:15:00 along the timelineId 2.

During recovery, PostgreSQL performs the following steps:

1. PostgreSQL reads the value of 'CHECKPOINT LOCATION' from the backup_label file.
2. Some values of parameters are read from the postgresql.conf (versions 11 or earlier, recovery.conf); in this example, restore_command, recovery_target_time and recovery_target_timeline.
3. PostgreSQL reads the timeline history file '00000002.history' that 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 '2024-1-1 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.

```
$ cat /home/postgres/archivelogs/00000003.history
1       0/A000198       before 2024-1-1 12:05:00.861324+00

2       0/B000078       before 2024-1-1 12:15:00.927133+00
```

For multiple PITR operations, the appropriate timelineId should be explicitly set to ensure the correct timeline history file is used.

In this way, timeline history files serve not only as logs of the database cluster's timeline history but also as recovery instruction documents for the PITR process.

10.5. INCREMENTAL BACKUP

In normal operation, backups need to be taken regularly. However, multiple full backups consume huge amounts of storage space.

To address this issue, PostgreSQL introduced incremental backups in version 17. An incremental backup saves only the changed portions of the files taken in the preceding backup.

In the following subsections, we will explore the overview of incremental backups, the process of creating incremental backups with pg_basebackup, and the data format of incremental backups.

10.5.1. Incremental Backup Overview

10.5.1.1. Taking Incremental Backup

Incremental backup is performed based on full backup and WAL summary files collected by WALsummarizer, described in Section 9.6.2. Therefore, we begin to explain incremental backup from taking a full backup.

1. Taking the full backup:
   After issuing do_pg_backup_start(), a full backup is taken, containing all relation files. We assume the REDO point of the full backup is $REDO_{full}$.
2. From full backup to incremental backup:
   The WAL Summarizer process tracks changes to all database blocks and writes these modifications to WAL summary files.
3. Taking the incremental backup:
   We assume the REDO point of this incremental backup is $REDO_{inc01}$. Using the summary files generated between $REDO_{full}$ and $REDO_{inc01}$, the incremental backup files, which contain only the changed blocks, are backed up instead of the entire relation files.

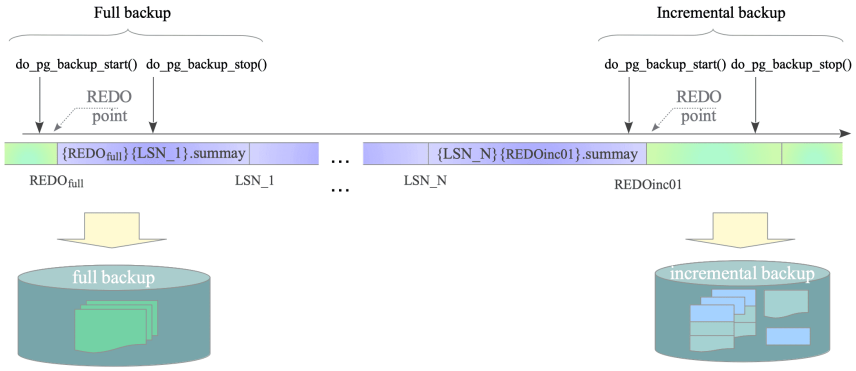

Figure 10.7. Concept of Taking an Incremental Backup.

Two important points to remember:

- In incremental backups, relation and visibility map files are backed up as incremental backup files, while free-space map files are always entirely backed up. This is because, as mentioned in Section 9.6.2, free-space map forks are not properly tracked by the WAL Summarizer.
- If relations are created after taking the preceding backup, the entire relation files are backed up.

Incremental backup files are named using the following convention:

```
INCREMENTAL.{oid}
INCREMENTAL.{oid}_vm
```

For instance, the incremental backup file for Table t1 (OID = 16551) is named 'INCREMENTAL.16551'.

Here's an example of the full and incremental backup files for t1:

```
$ ls -la -h backup/full/base/16425/ | grep "16551$"
-rw------- 1 postgres postgres 32K Oct 17 11:11 16551
16551
$ ls -la -h backup/inc01/base/16425/ | grep "16551$"
-rw------- 1 postgres postgres 24K Oct 17 11:20 INCREMENTAL.16551
INCREMENTAL.16551
```

The INCREMENTAL file format is described in Section 10.5.3.

10.5.1.2. Reconstructing Backup

To reconstruct a base backup from incremental backups, PostgreSQL provides pg_combinebackup utility.

Assuming that a full backup is located at '/usr/local/pgsql/backup/full' and an incremental backup at '/usr/local/pgsql/backup/inc01', we can reconstruct the base backup under '/usr/local/pgsql/reconstructed' by issuing the following command:

```
$ pg_combinebackup -d -n -o /usr/local/pgsql/reconstructed  \
>    /usr/local/pgsql/backup/full/  \
>    /usr/local/pgsql/backup/inc01/
```

Figure 10.8 illustrates how the pg_combinebackup utility reconstructs a base backup from a full backup and an incremental backup. In essence, pg_combinebackup applies the file changes stored in the incremental backup file to the original relation file.

For instance, assuming the relation file for Table t1 (OID=16551) is stored in the full backup and the incremental backup file named 'INCREMENTAL.16551' is in the incremental backup, pg_combinebackup overwrites the changed blocks (e.g., the 0th and 3rd blocks) stored in 'INCREMENTAL.16551' onto the relation file named '16551'.

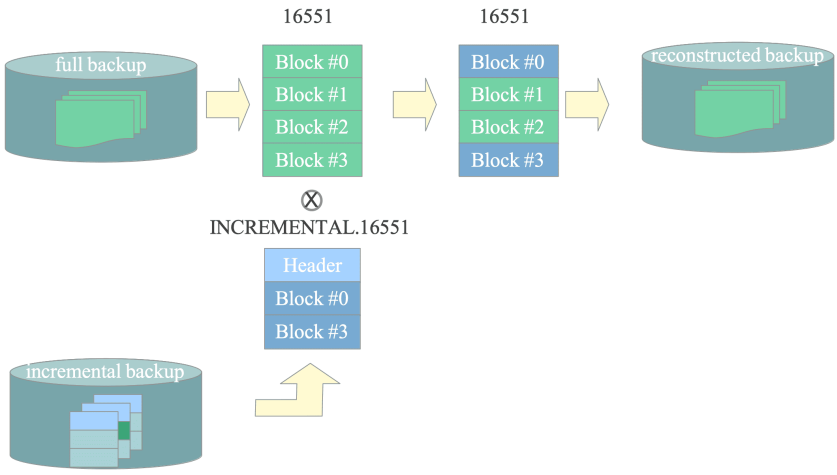

Figure 10.8. Concept of Reconstructing a Base Backup.

10.5.2. How to Take Incremental Backups

To take an incremental backup, use the '–incremental' option with the path to the preceding backup's manifest file.

For example, to take an incremental backup from host 192.168.1.10 to the local server at '/usr/local/pgsql/backup/inc01', based on the full backup located at '/usr/local/pgsql/backup/full', we can execute the following command:

```
$ pg_basebackup -h 192.168.1.10 -p 5432 \
>   --incremental /usr/local/pgsql/backup/full/backup_manifest \
>   -D /usr/local/pgsql/backup/inc01 -X stream -P -v
```

Figure 10.9 shows the sequence of how the pg_basebackup takes an incremental backup:

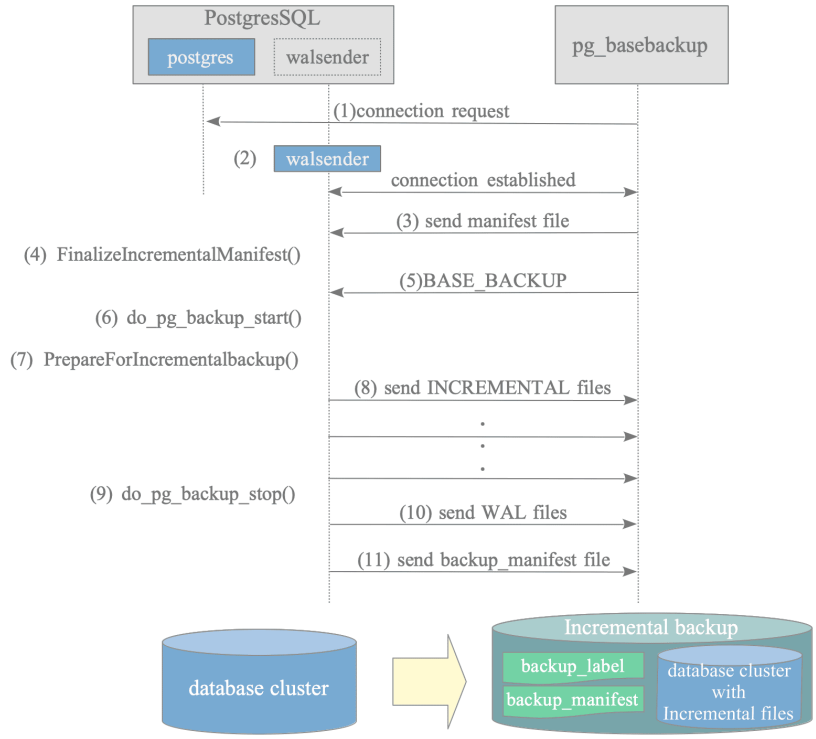

Figure 10.9. The sequence of the pg_basebackup in incremental backup mode.

1. Connection request:
2. Create walsender process:
3. Send backup manifest file:
   pg_basebackup sends the backup_manifest of the preceding backup.
4. Invoke FinalizeIncrementalManifest():
   The walsender process retrieves only two values, TimeLine and Start-LSN, from the backup manifest file.
5. Base backup request:
6. Do do_pg_backup_start():
7. Invoke PrepareForIncrementalbackup():
   Using the summary files from the Start-LSN retrieved in step 4 to the latest REDO point generated in step 6, the walsender process creates a list of changes to all relation blocks.
8. Send INCREMENTAL files:
   Using the change list created in step 7, the walsender process sends the incremental backup files, which are composed of a header block and the changed blocks, instead of the entire relation and visibility-map files.
9. Do do_pg_backup_stop():
10. Send WAL files:
11. Send backup_manifest file:

Note that, in step 8, the walsender sends the entire relation file if the relation is created after taking the preceding backup.

When taking the next incremental backup, we can execute the following command, setting the '–incremental' option to the previous incremental backup manifest '/usr/local/pgsql/backup/inc01/backup_manifest':

```
$ pg_basebackup -h 192.168.1.10 -p 5432 \
>   --incremental /usr/local/pgsql/backup/inc01/backup_manifest \
>   -D /usr/local/pgsql/backup/inc02 -X stream -P -v
```

10.5.3. Format of INCREMENTAL File

An INCREMENTAL file is typically composed of a header block and modified blocks, each block being 8 KB in size.

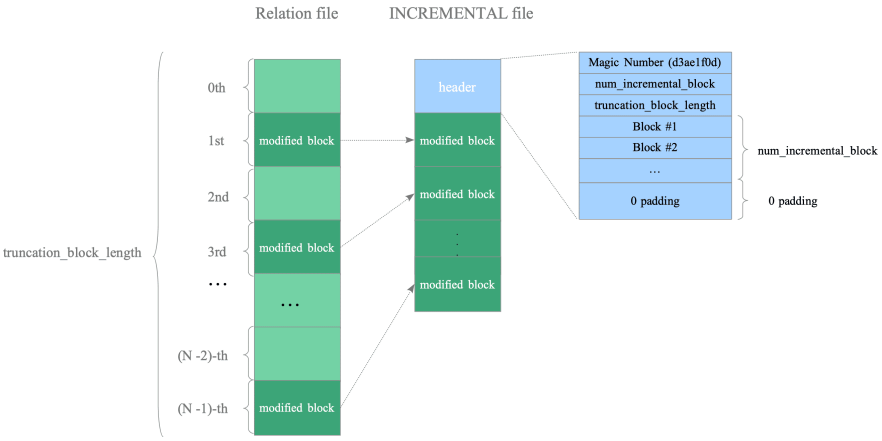

Figure 10.10. INCREMENTAL file format.

A header block contains four types of information:

- **Magic Number**: The header begins with "0xd3ae1f0d" to identify it as an incremental backup file.
- **num_incremental_block**: The number of modified blocks.
- **truncation_block_length**: In most cases, this is the total number of blocks in the table and VM that correspond to this file. However, depending on the conditions, it may be larger than this value due to internal processing. See the source code for details.
- **List of modified block numbers**: A list of modified block numbers. For instance, if 0th and 3rd blocks are modified, "0" and "3" are stored.

A header block is typically 8 KB in size, or a multiple of 8 KB. Normally, a header block is padded with zeros to reach an 8 KB alignment after the list of modified block numbers. If the list of modified block numbers exceeds 8 KB, additional header blocks are added.

If a table is truncated, dropped, or **unchanged**, its header block becomes a 12-byte block containing three fields: a magic number, a number of incremental blocks set to 0, and a truncation block length of 1. The INCREMENTAL file itself also becomes a 12-byte file containing only the header.

Figure 10.11 shows four examples of the INCREMENTAL files that are explained in Section 9.6.2.2:

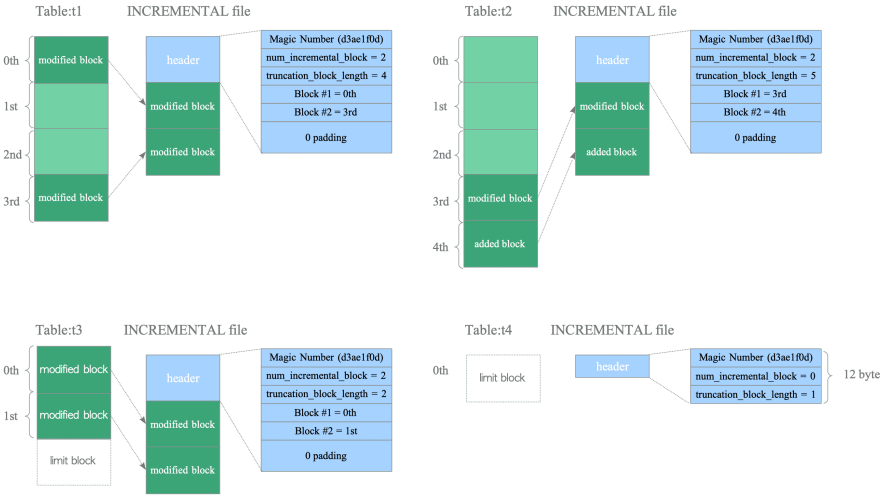

Figure 10.11. Four examples of the INCREMENTAL files

11. STREAMING REPLICATION

Synchronous 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 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.
- What is Replication Slots introduced in version 9.4.

Chapter Contents

Historical Info

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 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:

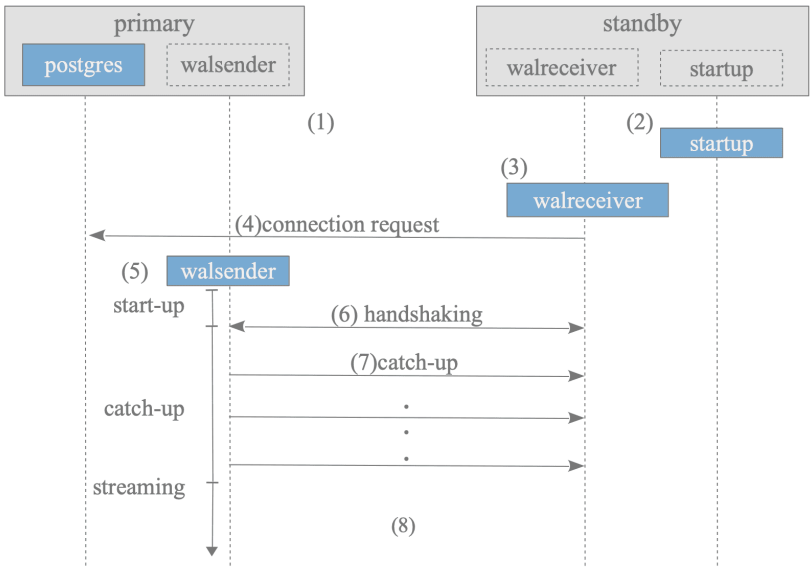

Figure 11.1. SR startup sequence.

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 Figures 11.1(5)–(6).
- catch-up – During the catch-up phase. See Figure 11.1(7).
- streaming – While Streaming Replication is working. See Figure 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.

11.1.1 What Happens When a Standby Server Restarts After a Long Downtime?

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_size (or wal_keep_segments in versions 12 or earlier) 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. Refer the Section 11.4 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 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 Figure 11.2:

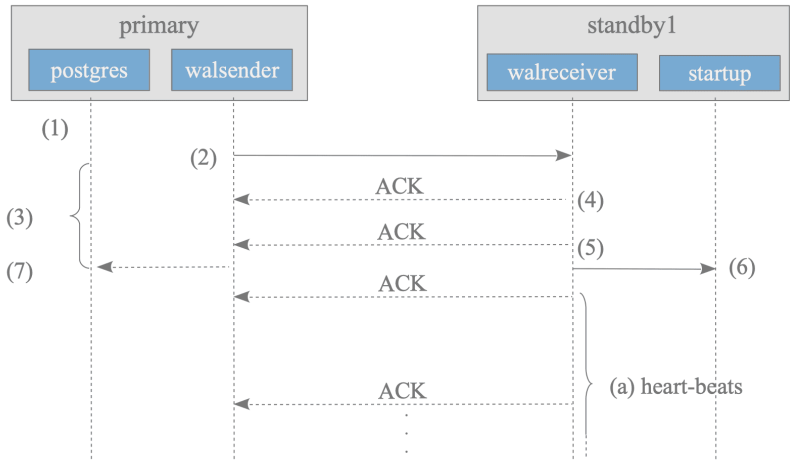

Figure 11.2. Streaming Replication's communication sequence diagram.

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:

- 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 be sent.

› XLogWalRcvSendReply

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)
```

Info

The heartbeat interval is set to the parameter wal_receiver_status_interval, which is 10 seconds by default.

11.2.2. 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.

11.2.3. Behavior When a Failure Occurs

This subsection describes 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.

```
$ 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.

11.2.4. Conflicts

In streaming replication, standbys can execute SELECT commands independently of the primary server. However, conflicts can arise between the standby and the primary server under certain conditions, potentially leading to errors.

Figure 11.3 shows a typical example: the primary server drops the table that the standby server is selecting.

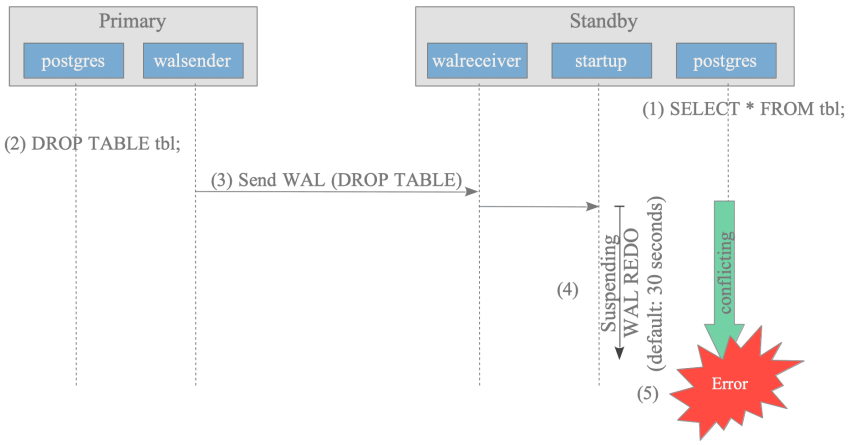

Figure 11.3. Conflict caused by DROP TABLE.

1. The standby server selects a table.
2. The primary server drops the table that the standby server is currently selecting.
3. The primary server sends XLOG records related to the DROP TABLE command.
4. The standby server suspends replaying the WAL data for the DROP TABLE command for 30 seconds by default (configurable with max_standby_archive_delay or max_standby_streaming_delay).
5. If the conflict is not resolved within the time specified by these parameters (i.e., the SELECT command is not completed), the SELECT command will return an error and terminate.

- Primary

```
testdb=# -- Primary


testdb=# DROP TABLE tbl;
DROP TABLE




testdb=#
```

- Standby

```
testdb=# -- Standby
testdb=# SELECT count(*) FROM tbl;









ERROR:  canceling statement due to conflict with
recovery
DETAIL:  User was holding a relation lock for
too long.
```

The cause of this conflict is the WAL data generated by the **Access Exclusive Lock** acquired internally by the DROP TABLE command on the primary[1], which conflicts with the standby's SELECT command. (As described in Section 9.5.1, WAL data includes not only changes to data but also lock information.)

According to the official document, the causes of conflicts can be classified into the following three types:

1. Access Exclusive Locks on the primary server: These conflict with any lock on the standby. Refer to the official document for a list of commands that acquire Access Exclusive Locks, including LOCK IN ACCESS EXCLUSIVE MODE, DROP TABLE, TRUNCATE, REINDEX, VACUUM FULL, etc.
2. Dropping databases or tablespaces.
3. Applying a vacuum cleanup record from WAL: If standby transactions can still see rows being removed or if queries are accessing the affected page.

Here is another example: the primary server deletes rows and performs the VACUUM command on a table being selected by the standby.

- Primary

```
testdb=# -- Primary


testdb=# DELETE FROM FROM tbl
testdb-#       WHERE data > 100000;
DELETE 1050
```

- Standby

```
testdb=# -- Standby
testdb=# SELECT count(*) FROM tbl;
```

```
testdb=# VACUUM tbl;
VACUUM



testdb=#
```

```
ERROR:  canceling statement due to conflict with
recovery
DETAIL:  User query might have needed to see row
versions that must be removed.
```

Conflicts caused by VACUUM processing are particularly troublesome because they can occur not only during explicitly run VACUUM commands but also during autovacuum operations, which is described in Section 6.5.

11.2.4.1. Mitigating Conflicts Caused by Locking

On standby servers, conflicts can be mitigated by increasing the values of max_standby_archive_delay and max_standby_streaming_delay (default: 30 seconds). These settings allow the standby to delay replaying WAL data, reducing the likelihood of conflicts.

However, these parameters cannot always eliminate conflicts.

Additionally, they affect other PostgreSQL backends on the standby, preventing them from accessing the latest data during the delay. This means the standby server is not fully synchronous during such conflicts.

It is essential to configure these parameters while carefully considering the disadvantages of the settings and the operational trade-offs.

11.2.4.2. Avoiding Conflicts Caused by Vacuum Processing

On the primary server, by setting the parameter hot_standby_feedback (default: off) to `on`, the primary delays deleting data that the standby needs during vacuum processing, based on the standby's state. This helps avoid conflicts caused by vacuum operations.

The standby's state is sent to the primary at intervals defined by wal_receiver_status_interval (default: 10 seconds).

While enabling *hot_standby_feedback* resolves vacuum-induced conflicts, it has the following drawbacks for the primary server:

- Table and Index Bloat: Retaining old tuples visible to the standby prevents vacuuming, leading to increased table and index bloat on the primary.
- WAL Retention Issues: Retaining additional data for standby consistency can increase WAL usage.

pg_stat_database_conflicts

Issuing the pg_stat_database_conflicts on a standby server displays the causes and number of conflicts:

```
testdb=# -- Standby
testdb=# \x
Expanded display is on.
testdb=# SELECT * FROM pg_stat_database_conflicts WHERE datname = 'testdb';
-[ RECORD 1 ]------------+-------
datid                    | 16384
datname                  | testdb
confl_tablespace         | 0
confl_lock               | 1
confl_snapshot           | 1
confl_bufferpin          | 0
confl_deadlock           | 0
confl_active_logicalslot | 0
```

Historical Info

Until version 15, the vacuum_defer_cleanup_age parameter was supported to defer the deletion of dead tuples during a vacuum operation. If a positive number was set, vacuum operations would defer deleting the dead tuples for the specified number of transactions.

This parameter was removed in version 16 because it could not always eliminate conflicts, and using hot_standby_feedback and Replication Slots, described in Section 11.4, could manage conflicts more effectively.

---

1. More precisely, in situations where the primary drops objects that the standby is accessing, streaming replication is designed to transition to a conflict state. This serves as a grace period to prevent the standby from suddenly returning an error.
   As part of this design, the acquisition of Access Exclusive Locks is recorded as XLOG records. Note that Access Exclusive Locks do not generate XLOG records if wal_level is set to minimal. That is, XLOG records for these locks are required for replication, not for ordinary recovery.
   Therefore, this type of conflict is not caused accidentally by the Access Exclusive Lock itself but is intentionally introduced to avoid sudden errors on the standby. ↩

11.3. MANAGING MULTIPLE-STANDBY SERVERS

This section describes how 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.)
- **quorum:** The standby servers are in quorum mode. See Section 11.3.2.1 for detail.

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 11.4 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.

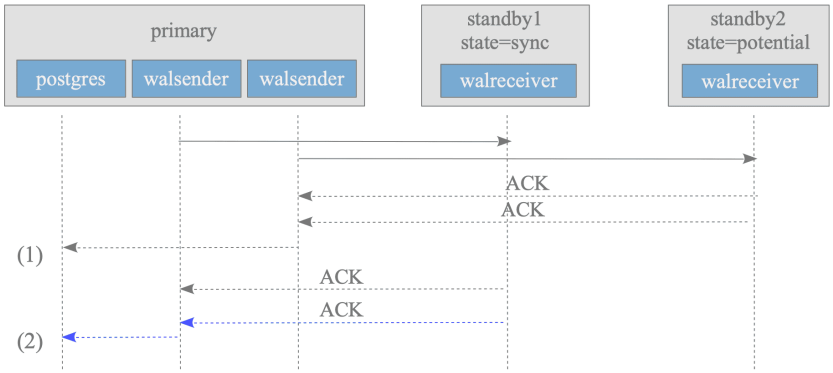

Figure 11.4. Managing multiple standby servers.

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.

11.3.2.1. Quorum-Based Synchronous Replication

In version 9.6, quorum-based synchronous replication was introduced. This feature allows transactions in PostgreSQL to be considered committed once a subset (a quorum) of synchronous standby servers acknowledges them. It enhances flexibility in RDBMS configuration design and administration.

Quorum-based synchronous replication has two modes: ANY and FIRST.

ANY Mode

The format of synchronous_standby_names for ANY mode is:

```
ANY num_sync ( standby_name [, ...] )
```

In ANY mode, the primary server completes the commit action of the current transaction any 'num_sync' standby servers in the list return ACK responses.

For example, the following setting allows the primary server to complete the commit action of the current transaction as soon as any two standby servers return ACK responses, even if the rest of the standby servers do not return responses.

```
synchronous_standby_names = 'ANY 2 (standby1, standby2, standby3)'
```

Figure 11.5 illustrates the behavior of the ANY Mode setting:

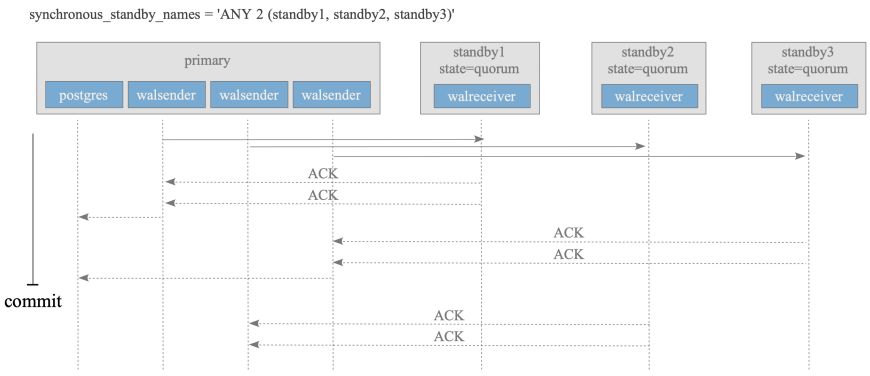

Figure 11.5. Behavior of the ANY Mode Setting.

The following is the state of sync_priority and sync_state for the above setting:

```
testdb=# SELECT application_name AS host,
        sync_priority, sync_state FROM pg_stat_replication;
   host   | sync_priority | sync_state
----------+---------------+------------
 standby1 |             1 | quorum
 standby2 |             1 | quorum
 standby3 |             1 | quorum
(3 rows)
```

FIRST Mode

The format of synchronous_standby_names for FIRST mode is:

```
[FIRST] num_sync ( standby_name [, ...] )
```

In FIRST mode, the primary server completes the commit action of the current transaction after the first 'num_sync' standby servers in the list return ACK responses.

For example, the following setting allows the primary server to wait for the commit action of the current transaction until the first two standby servers, i.e., standby1 and standby2, return ACK responses, even if standby3 has already returned ACK response:

```
synchronous_standby_names = 'FIRST 2 (standby1, standby2, standby3)'
```

Figure 11.6 illustrates the behavior of the FIRST Mode setting:

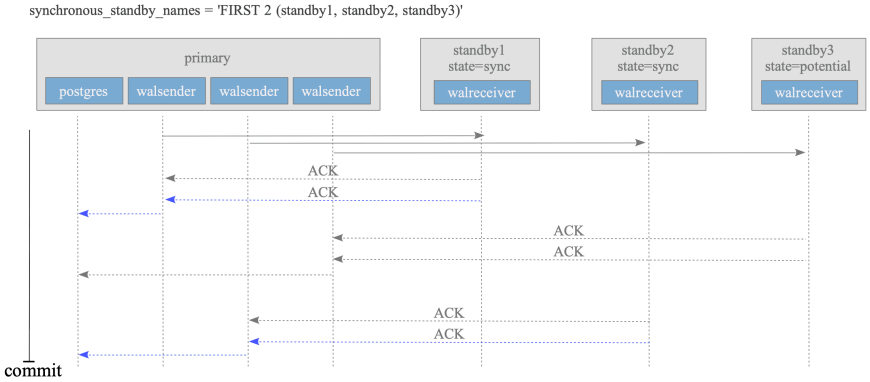

Figure 11.6. Behavior of the FIRST Mode Setting.

The following is the state of sync_priority and sync_state for the above setting:

```
SELECT application_name AS host,
        sync_priority, sync_state FROM pg_stat_replication;
   host   | sync_priority | sync_state
----------+---------------+------------
 standby3 |             3 | potential
 standby2 |             2 | sync
 standby1 |             1 | sync
(3 rows)
```

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 Figure 11.7. 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.)

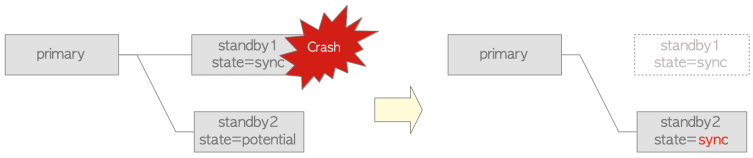

Figure 11.7. Replacing of synchronous standby server.

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. REPLICATION SLOTS

As discussed in Section 11.1.1, Replication slots, introduced in version 9.4, were designed primarily to ensure that WAL segments and old tuple versions are retained long enough to support replication completion.

We will explore replication slots in this section.

Note that although replication slots are fundamental to logical replication, this section does not cover them in that context.

11.4.1. Advantages of Replication Slots in Streaming Replication

In streaming replication, although replication slots are not mandatory, they offer the following advantages compared to using wal_keep_size:

1. Ensure Streaming Replication Works Without Losing Required WAL Segments:
   Replication slots track which WAL segments are needed and prevent their removal. In contrast, when using only wal_keep_size, necessary segments may be removed if standbys do not read them for an extended period.

2. Maintain Only the Minimum Necessary WAL Segments:
   With replication slots, only the required WAL segments are kept in the pg_wal directory, while unnecessary segments are removed. Conversely, using wal_keep_size keeps a fixed number of WAL segments, regardless of whether they are needed or not.

max_slot_wal_keep_size

Since Replication Slots can store WAL segments indefinitely, there is a risk of the storage area filling up with stored WAL segments in the worst-case scenario, potentially leading to an operating system panic.

To address this issue, the configuration parameter max_slot_wal_keep_size was introduced in version 13. It limits the maximum size of WAL segments in the pg_wal directory at checkpoint time.

The key difference between using max_slot_wal_keep_size with replication slots and wal_keep_size is in how they manage WAL segments:

- max_slot_wal_keep_size sets a maximum size limit for WAL segments while allowing replication slots to retain only the minimum required amount.
- wal_keep_size specifies a fixed number of WAL segments to be retained, regardless of whether they are needed or not.

11.4.2. Replication Slots and Related Processes and Files

Replication Slots are stored in the memory area allocated in the shared memory.

Figure 11.8 illustrates replication slots and the related processes and files:

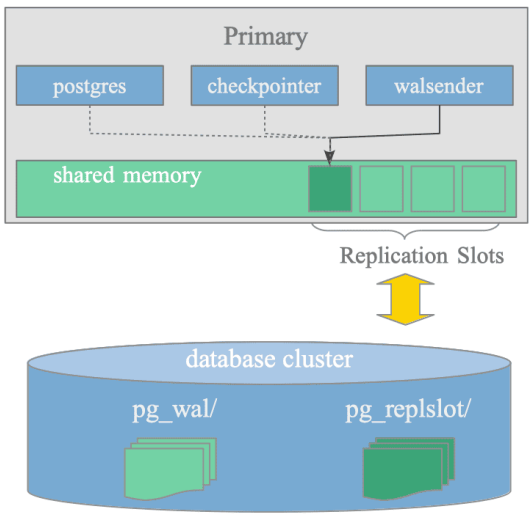

Figure 11.8. Replication Slots and Related Processes and Files.

Related Processes:

The processes related to Replication Slots are shown below:

- **Walsender**: Continuously updates the corresponding replication slot to reflect the current state of WAL data of the standby server.
- **Checkpointer background worker**: Reads the replication slots to determine whether WAL segments can be removed or not during checkpointing.
- **Postgres backend**: Displays the slot information using the system view pg_replication_slots.

Related Files:

The files related to Replication Slots are shown below:

- **State files** under the *pg_replslot* directory: Walsenders regularly save detailed information about their replication slots to state files located in this directory. When the server is restarted, it loads this saved information back into memory to restore the status of its replication slots.
  The state is defined by the `ReplicationSlotPersistentData` structure, described in the next section.
- **WAL segment files** under the *pg_wal* directory: The number of WAL segment files are managed by the checkpointer background worker. The checkpointer prioritizes Replication Slots and wal_keep_size over max_wal_size to ensure essential data is not removed.

11.4.3. Data Structure

Replication Slots are defined by the `ReplicationSlot` structure in slot.h.

ReplicationSlot

```
/*
 * Shared memory state of a single replication slot.
 *
 * The in-memory data of replication slots follows a locking model based
 * on two linked concepts:
 * - A replication slot's in_use flag is switched when added or discarded using
 * the LWLock ReplicationSlotControlLock, which needs to be hold in exclusive
 * mode when updating the flag by the backend owning the slot and doing the
 * operation, while readers (concurrent backends not owning the slot) need
 * to hold it in shared mode when looking at replication slot data.
 * - Individual fields are protected by mutex where only the backend owning
 * the slot is authorized to update the fields from its own slot.  The
 * backend owning the slot does not need to take this lock when reading its
 * own fields, while concurrent backends not owning this slot should take the
 * lock when reading this slot's data.
 */
typedef struct ReplicationSlot
{
    /* lock, on same cacheline as effective_xmin */
    slock_t     mutex;

    /* is this slot defined */
    bool        in_use;

    /* Who is streaming out changes for this slot? 0 in unused slots. */
    pid_t       active_pid;

    /* any outstanding modifications? */
    bool        just_dirtied;
    bool        dirty;

    /*
     * For logical decoding, it's extremely important that we never remove any
     * data that's still needed for decoding purposes, even after a crash;
     * otherwise, decoding will produce wrong answers.  Ordinary streaming
     * replication also needs to prevent old row versions from being removed
     * too soon, but the worst consequence we might encounter there is
     * unwanted query cancellations on the standby.  Thus, for logical
     * decoding, this value represents the latest xmin that has actually been
     * written to disk, whereas for streaming replication, it's just the same
     * as the persistent value (data.xmin).
     */
    TransactionId effective_xmin;
    TransactionId effective_catalog_xmin;

    /* data surviving shutdowns and crashes */
    ReplicationSlotPersistentData data;

    /* is somebody performing io on this slot? */
    LWLock      io_in_progress_lock;

    /* Condition variable signaled when active_pid changes */
    ConditionVariable active_cv;

    /* all the remaining data is only used for logical slots */

    /*
     * When the client has confirmed flushes >= candidate_xmin_lsn we can
     * advance the catalog xmin.  When restart_valid has been passed,
     * restart_lsn can be increased.
     */
    TransactionId candidate_catalog_xmin;
    XLogRecPtr  candidate_xmin_lsn;
    XLogRecPtr  candidate_restart_valid;
    XLogRecPtr  candidate_restart_lsn;
```

```
    /*
     * This value tracks the last confirmed_flush LSN flushed which is used
     * during a shutdown checkpoint to decide if logical's slot data should be
     * forcibly flushed or not.
     */
    XLogRecPtr  last_saved_confirmed_flush;

    /* The time since the slot has become inactive */
    TimestampTz inactive_since;
} ReplicationSlot;

#define SlotIsPhysical(slot) ((slot)->data.database == InvalidOid)
#define SlotIsLogical(slot) ((slot)->data.database != InvalidOid)

/*
 * Shared memory control area for all of replication slots.
 */
typedef struct ReplicationSlotCtlData
{
    /*
     * This array should be declared [FLEXIBLE_ARRAY_MEMBER], but for some
     * reason you can't do that in an otherwise-empty struct.
     */
    ReplicationSlot replication_slots[1];
} ReplicationSlotCtlData;
```

Although the structure contains many items, as it is shared between both streaming and logical replication, the main items relevant to streaming replication are as follows:

- pid_t **active_pid**: The PID of the walsender process that manages this slot.
- ReplicationSlotPersistentData **data**: Items defined by `ReplicationSlotPersistentData` structure. The main items include:
    - NameData **name**: The name of the slot.
    - XLogRecPtr **restart_lsn**: The oldest LSN that might be required by this replication slot. The checkpointer reads the minimum restart_lsn value across all slots to determine whether WAL segments can be removed.

ReplicationSlotPersistentData

```
/*
 * On-Disk data of a replication slot, preserved across restarts.
 */
typedef struct ReplicationSlotPersistentData
{

    NameData    name;

    /* database the slot is active on */
    Oid         database;

    /*
     * The slot's behaviour when being dropped (or restored after a crash).
     */
    ReplicationSlotPersistency persistency;

    TransactionId xmin;

    /*
     * xmin horizon for catalog tuples
     *
     * NB: This may represent a value that hasn't been written to disk yet;
     * see notes for effective_xmin, below.
     */
    TransactionId catalog_xmin;

    /* oldest LSN that might be required by this replication slot */
    XLogRecPtr  restart_lsn;

    /* RS_INVAL_NONE if valid, or the reason for having been invalidated */
    ReplicationSlotInvalidationCause invalidated;

    /*
     * Oldest LSN that the client has acked receipt for.  This is used as the
     * start_lsn point in case the client doesn't specify one, and also as a
     * safety measure to jump forwards in case the client specifies a
     * start_lsn that's further in the past than this value.
     */
    XLogRecPtr  confirmed_flush;

    /*
     * LSN at which we enabled two_phase commit for this slot or LSN at which
     * we found a consistent point at the time of slot creation.
     */
    XLogRecPtr  two_phase_at;

    /*
     * Allow decoding of prepared transactions?
     */
    bool        two_phase;

    /* plugin name */
    NameData    plugin;

    /*
     * Was this slot synchronized from the primary server?
     */
    char        synced;

    /*
     * Is this a failover slot (sync candidate for standbys)? Only relevant
     * for logical slots on the primary server.
     */
```

```
    bool        failover;
} ReplicationSlotPersistentData;
```

The data stored in *ReplicationSlotPersistentData structure* is regularly saved in the pg_replslot directory.

11.4.4. Starting Replication Slot

Figure 11.9 illustrates the starting sequence of a replication slot:

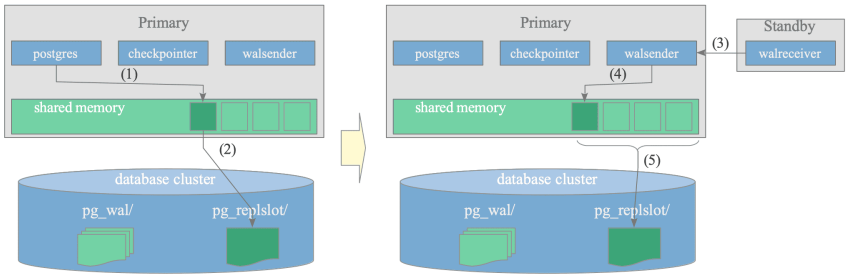

Figure 11.9. Starting Sequence of a Replication Slot.

1. Creating a (physical) replication slot using the pg_create_physical_replication_slot() function.
   Except for the slot name, the data written to the replication slot is set to its default value.

   ```
   testdb=# SELECT * FROM pg_create_physical_replication_slot('standby_slot');
      slot_name    | lsn
   ---------------+-----
    standby_slot  |
   (1 row)
   ```

2. Writing a portion of the slot data, defined by the *ReplicationSlotPersistentData* structure, in the pg_replslot directory.
   A file named *state* is created under the subdirectory corresponding to the slot name, as shown below:

   ```
   $ ls -1 pg_replslot/
   standby_slot
   $ find pg_replslot/
   pg_replslot/
   pg_replslot/standby_slot
   pg_replslot/standby_slot/state
   ```

3. (Re)Connecting standby server to the primary server.
   To (re)connect the standby server to the primary server, set the primary_slot_name configuration parameter to the name of the replication slot.

   ```
   # standby's postgresql.conf
   primary_slot_name = 'standby_slot'
   ```

   Then, issue the pg_ctl command with the reload option:

   ```
   $ pg_ctl -D $PGDATA_STANDBY reload
   ```

4. Updating the replication slot, e.g., active_pid, restart_lsn, etc.

5. Writing a portion of the updated slot data in the pg_replslot directory.

11.4.5. Managing Replication Slots

After replication slots are set in shared memory, walsender processes continuously update the slots to reflect the current states of the corresponding standby servers.

Below is an example of the states of the replication slots:

```
testdb=# \x
Expanded display is on.
testdb=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-------+--------------
slot_name           | standby_slot
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 236772
```

```
xmin                | 754
catalog_xmin        |
restart_lsn         | 0/303B968
confirmed_flush_lsn |
wal_status          | reserved
safe_wal_size       |
two_phase           | f
inactive_since      |
conflicting         |
invalidation_reason |
failover            | f
synced              | f
```

The primary PostgreSQL server regularly saves detailed information about its replication slots to state files in the pg_replslot directory. (When the primary server restarts, it loads this saved information back into memory to restore the status of its replication slots.)