Chapter 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.

Section 6.1 outlines the concurrent VACUUM process. Then, subsequent sections describe the following.

- Visibility Map
- Freeze processing
- Removing unnecessary clog files
- Autovacuum daemon
- Full VACUUM

6.1. Outline of Concurrent VACUUM

Vacuum processing performs the following tasks for specified tables or all tables in the database:

1. Removing dead tuples
   - Remove dead tuples and defragment live tuples for each page.
   - Remove index tuples that point to dead tuples.
2. Freezing old txids
   - Freeze old txids of tuples if necessary.
   - Update frozen txid related system catalogs (pg_database and pg_class).
   - Remove unnecessary parts of the clog if possible.
3. Others
   - Update the FSM and VM of processed tables.
   - Update several statistics (pg_stat_all_tables, etc).

It is assumed that readers are familiar with following terms: dead tuples, freezing txid, FSM, and the clog; if you are not, refer to Chapter 5. VM is introduced in Section 6.2.

The following pseudocode describes vacuum processing.

</> Pseudocode: Concurrent VACUUM

```
(1)  FOR each table
(2)      Acquire a ShareUpdateExclusiveLock lock for the target table

         /* The first block */
(3)      Scan all pages to get all dead tuples, and freeze old tuples if necessary
(4)      Remove the index tuples that point to the respective dead tuples if exists

         /* The second block */
(5)      FOR each page of the table
(6)          Remove the dead tuples, and Reallocate the live tuples in the page
(7)          Update FSM and VM
         END FOR

         /* The third block */
(8)      Clean up indexes
(9)      Truncate the last page if possible
(10      Update both the statistics and system catalogs of the target table
         Release the ShareUpdateExclusiveLock lock
     END FOR

     /* Post-processing */
(11)  Update statistics and system catalogs
(12)  Remove both unnecessary files and pages of the clog if possible
```

(1) Get each table from the specified tables.
(2) Acquire a ShareUpdateExclusiveLock lock for the table. This lock allows reading from other transactions.
(3) Scan all pages to get all dead tuples, and freeze old tuples if necessary.
(4) Remove the index tuples that point to the respective dead tuples if exists.
(5) Do the following tasks, step (6) and (7), for each page of the table.
(6) Remove the dead tuples and Reallocate the live tuples in the page.
(7) Update both the respective FSM and VM of the target table.
(8) Clean up the indexes by the index_vacuum_cleanup()@indexam.c function.
(9) Truncate the last page if the last one does not have any tuple.
(10) Update both the statistics and the system catalogs related to vacuum processing for the target table.
(11) Update both the statistics and the system catalogs related to vacuum processing.
(12) Remove both unnecessary files and pages of the clog if possible.

This pseudocode has two sections: a loop for each table and post-processing. The inner loop can be divided into three blocks. Each block has individual tasks.

These three blocks and the post-process are outlined in the following.

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.

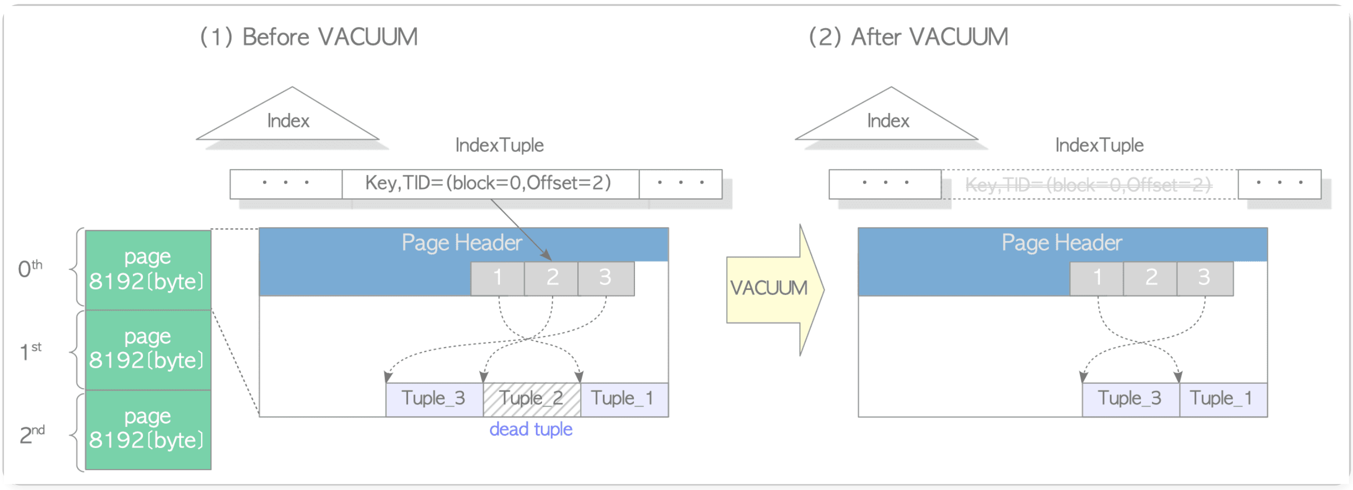
After scanning, PostgreSQL removes index tuples by referring to the dead tuple list. This process is internally called the "cleanup stage". It is a costly process, so PostgreSQL was improved in verion 11. In versions 10 or earlier, the cleanup stage is always executed. In versions 11 or later, if the target index is B-tree, whether the cleanup stage is executed or not is decided by the configuration parameter vacuum_cleanup_index_scale_factor. See the description of this parameter in details.

If maintenance_work_mem is full and scanning is incomplete, PostgreSQL proceeds to the next tasks, i.e. steps (4) to (7). Then, it goes back to step (3) and proceeds remainder scanning.

6.1.2. Second Block

This block removes dead tuples and updates both the FSM and VM on a page-by-page basis. Figure 6.1 shows an example:

Fig. 6.1. Removing a dead tuple.



Assume that the table contains three pages. We focus on the 0th page (i.e., the first page). This page has three tuples. Tuple_2 is a dead tuple (Fig. 6.1(1)). In this case, PostgreSQL removes Tuple 2 and reorders the remaining tuples to repair fragmentation. Then, it updates both the FSM and VM of this page (Fig. 6.1(2)). PostgreSQL continues this process until the last page.

Note that unnecessary line pointers are not removed. They will be reused in the future. This is because if line pointers are removed, all index tuples of the associated indexes must be updated.

6.1.3. Third Block

The third block performs the cleanup after the deletion of the indexes, and also updates the statistics and system catalogs related to vacuum processing for each target table.

Moreover, if the last page has no tuples, it is truncated from the table file.

6.1.4. Post-processing

When vacuum processing is complete, PostgreSQL updates all the statistics and system catalogs related to vacuum processing. It also removes unnecessary parts of the clog if possible (Section 6.4).

Ring Buffer

Vacuum processing uses a *ring buffer*, described in Section 8.5. Therefore, processed pages are not cached in the shared buffers.

6.2. Visibility Map

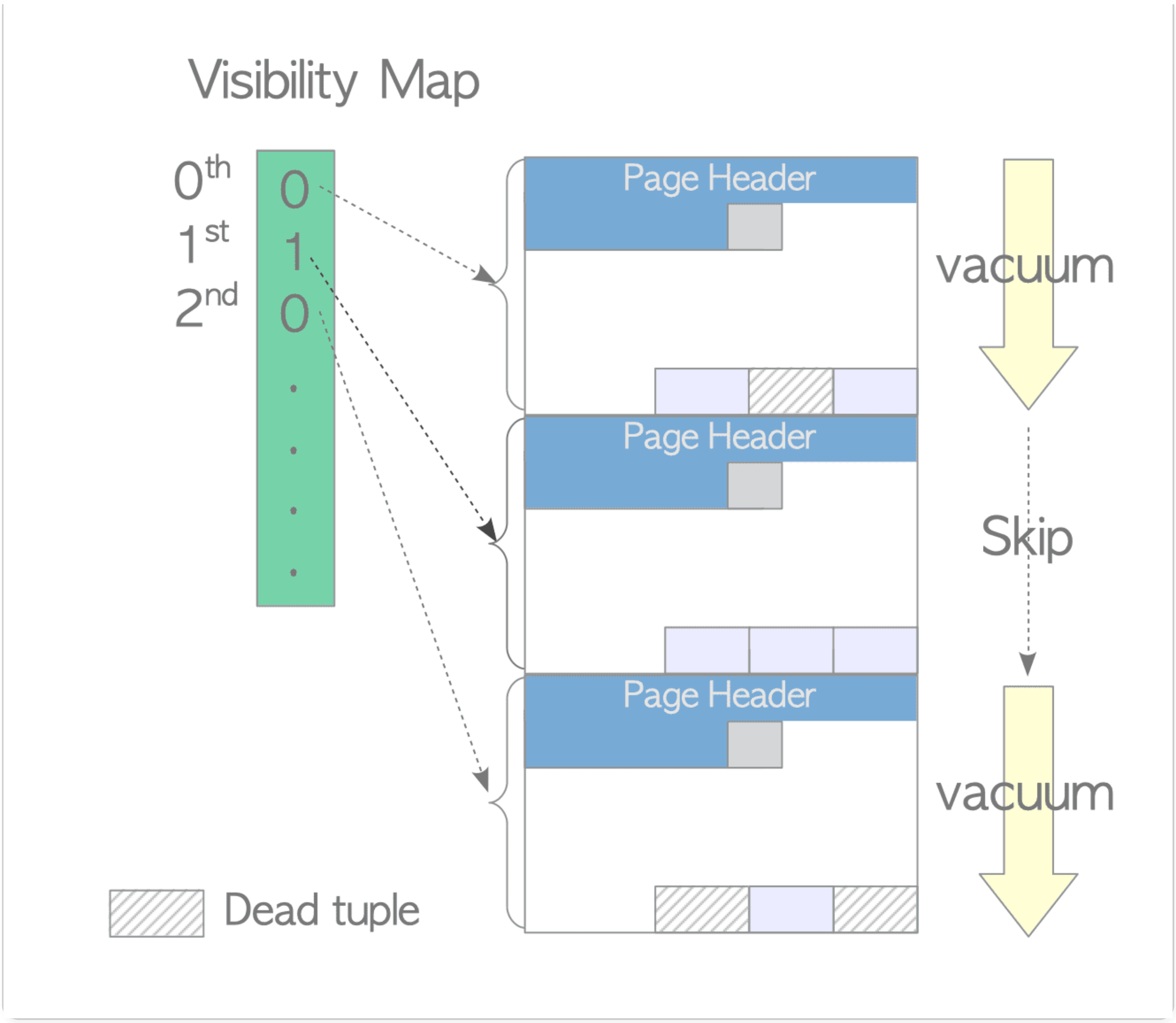
Vacuum processing is costly. Therefore, the VM was introduced in version 8.4 to reduce this cost.

The basic concept of the VM is simple. Each table has an individual visibility map that holds the visibility of each page in the table file. The visibility of pages determines whether or not each page has dead tuples. Vacuum processing can skip a page that does not have dead tuples by using the corresponding visibility map (VM).

Figure 6.2 shows how the VM is used. Suppose that the table consists of three pages, and the 0th and 2nd pages contain dead tuples and the 1st page does not. The VM of this table holds information about which pages contain dead tuples. In this case, vacuum processing skips the 1st page by referring to the VM's information.

Each VM is composed of one or more 8 KB pages, and this file is stored with the 'vm' suffix. As an example, one table file whose relfilenode is 18751 with FSM (18751_fsm) and VM (18751_vm) files shown in the following.

Fig. 6.2. How the VM is used.



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

6.2.1. Enhancement of VM

The VM was enhanced in version 9.6 to improve the efficiency of freeze processing. The new VM shows page visibility and information about whether tuples are frozen or not in each page (Section 6.3.3).

6.3. Freeze Processing

Freeze processing has two modes. For convenience, these modes are referred to as **lazy mode** and **eager mode**. It is performed in either mode depending on certain conditions.

Concurrent VACUUM is often called "lazy vacuum" internally. However, the lazy mode defined in this document is a mode of freeze processing.

Freeze processing typically runs in lazy mode, but eager mode is run when specific conditions are satisfied.

In lazy mode, freeze processing scans only pages that contain dead tuples using the respective VM of the target tables.

In contrast, eager mode scans all pages regardless of whether each page contains dead tuples or not. It also updates system catalogs related to freeze processing and removes unnecessary parts of the clog if possible.

Sections 6.3.1 and 6.3.2 describe these modes, respectively. Section 6.3.3 describes how to improve the freeze process in eager mode.

6.3.1. Lazy Mode

When starting freeze processing, PostgreSQL calculates the freezeLimit txid and freezes tuples whose t_xmin is less than the freezeLimit txid.

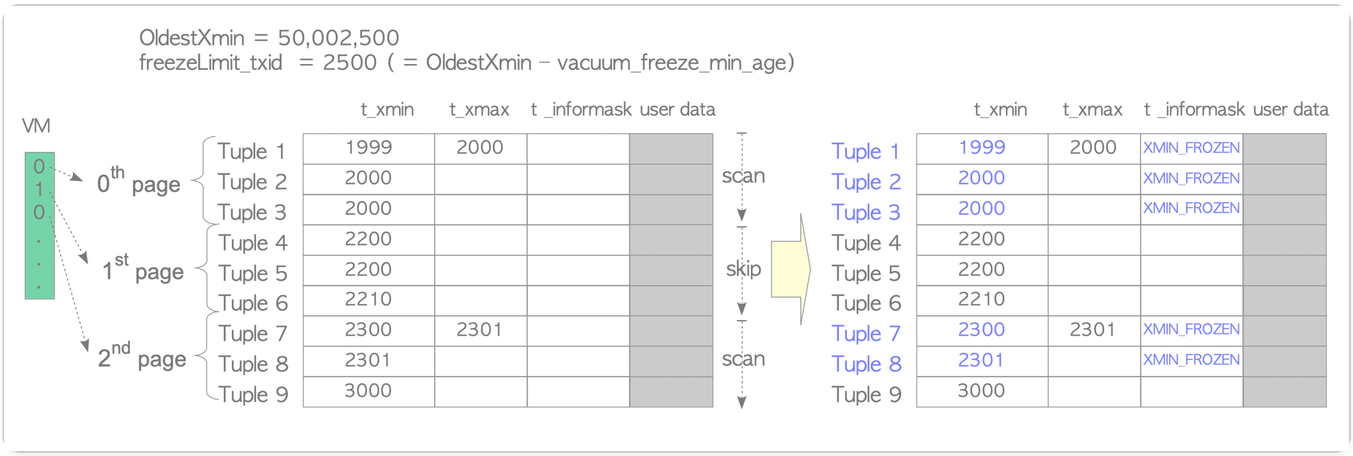
The freezeLimit txid is defined as follows:

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

where `OldestXmin` is the oldest txid among currently running transactions. For example, if three transactions (txids 100, 101, and 102) are running when the VACUUM command is executed, `OldestXmin` is 100. If no other transactions exist, `OldestXmin` is the txid that executes this VACUUM command. Here, vacuum_freeze_min_age is a configuration parameter (the default is 50,000,000).

Figure 6.3 shows a specific example. Here, Table_1 consists of three pages, and each page has three tuples. When the VACUUM command is executed, the current txid is 50,002,500 and there are no other transactions. In this case, `OldestXmin` is 50,002,500; thus, the freezeLimit txid is 2500. Freeze processing is executed as follows.

Fig. 6.3. Freezing tuples in lazy mode.



	t_xmin	t_xmax	t_informask	user data
Tuple 1	1999	2000		
Tuple 2	2000			
Tuple 3	2000			
Tuple 4	2200			
Tuple 5	2200			
Tuple 6	2210			
Tuple 7	2300	2301		
Tuple 8	2301			
Tuple 9	3000			

	t_xmin	t_xmax	t_informask	user data
Tuple 1	1999	2000	XMIN_FROZEN	
Tuple 2	2000		XMIN_FROZEN	
Tuple 3	2000		XMIN_FROZEN	
Tuple 4	2200			
Tuple 5	2200			
Tuple 6	2210			
Tuple 7	2300	2301	XMIN_FROZEN	
Tuple 8	2301		XMIN_FROZEN	
Tuple 9	3000			

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:

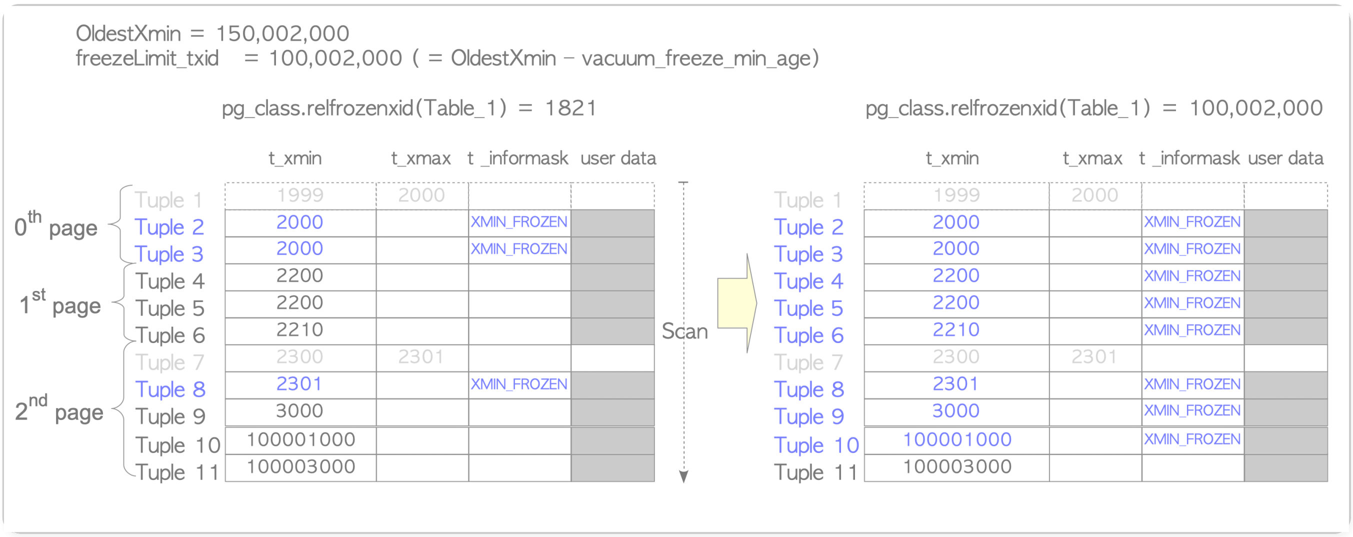
$$\texttt{pg\_database.datfrozenxid} < (\texttt{OldestXmin} - \texttt{vacuum\_freeze\_table\_age})$$

In the condition above, *pg_database.datfrozenxid* represents the columns of the pg_database system catalog and holds the oldest frozen txid for each database. Details are described later; therefore, we assume that the value of all pg_database.datfrozenxid are 1821 (which is the initial value just after installation of a new database cluster in version 9.5). Vacuum_freeze_table_age is a configuration parameter (the default is 150,000,000).

Figure 6.4 shows a specific example. In Table_1, both Tuple_1 and Tuple_7 have been removed. Tuple_10 and Tuple_11 have been inserted into the 2nd page. When the VACUUM command is executed, the current txid is 150,002,000, and there are no other transactions. Thus, OldestXmin is 150,002,000 and the freezeLimit txid is 100,002,000. In this case, the above condition is satisfied because '$1821 < (150002000 - 150000000)$'; therefore, the freeze processing performs in eager mode as follows.

(Note that this is the behavior of versions 9.5 or earlier; the latest behavior is described in Section 6.3.3.)

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



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.

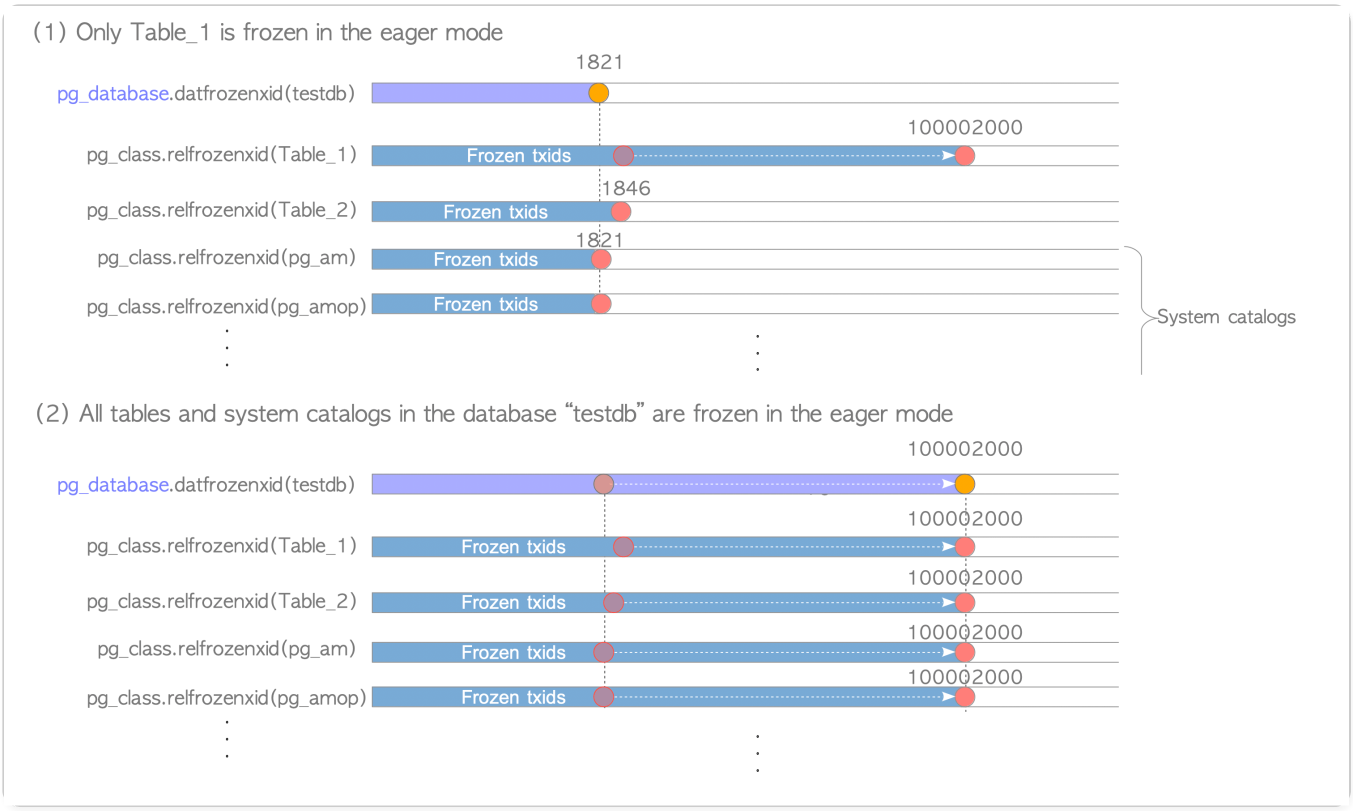
2nd page:

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 (Fig. 6.5(1)). If all relations in the current database are frozen in eager mode, the pg_database.datfrozenxid of the database is updated because all relations' pg_class.relfrozenxid for this database are updated to the current freezeLimit txid (Fig. 6.5(2)).

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



(1) Only Table_1 is frozen in the eager mode
1821
pg_database.datfrozenxid(testdb)
100002000
pg_class.relfrozenxid(Table_1)
Frozen txids
1846
pg_class.relfrozenxid(Table_2)
Frozen txids
1821
pg_class.relfrozenxid(pg_am)
Frozen txids
pg_class.relfrozenxid(pg_amop)
Frozen txids
System catalogs
(2) All tables and system catalogs in the database "testdb" are frozen in the eager mode
100002000
pg_database.datfrozenxid(testdb)
100002000
pg_class.relfrozenxid(Table_1)
Frozen txids
100002000
pg_class.relfrozenxid(Table_2)
Frozen txids
100002000
pg_class.relfrozenxid(pg_am)
Frozen txids
100002000
pg_class.relfrozenxid(pg_amop)
Frozen txids

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.

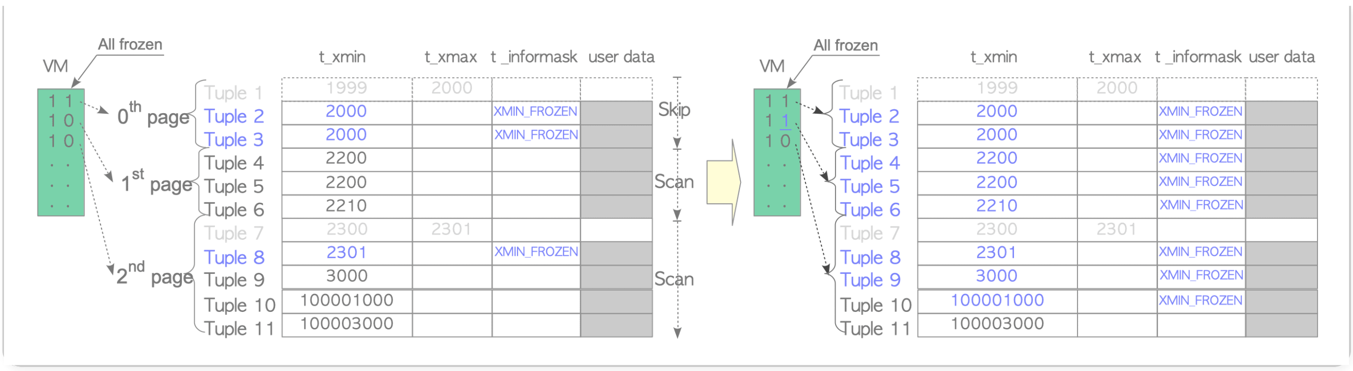
6.3.3. Improving Freeze Processing in Eager Mode

The eager mode in version 9.5 or earlier versions is not efficient because always scans all pages. For instance, in the example of Section 6.3.2, the 0th page is scanned even though all tuples in its page are frozen.

To deal with this issue, the VM and freeze process have been improved in version 9.6. As mentioned in Section 6.2.1, the new VM has information about whether all tuples are frozen in each page. When freeze processing is executed in eager mode, pages that contain only frozen tuples can be skipped.

Figure 6.6 shows an example. When freezing this table, the 0th page is skipped by referring to the VM's information. After freezing the 1st page, the associated VM information is updated because all tuples of this page have been frozen.

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

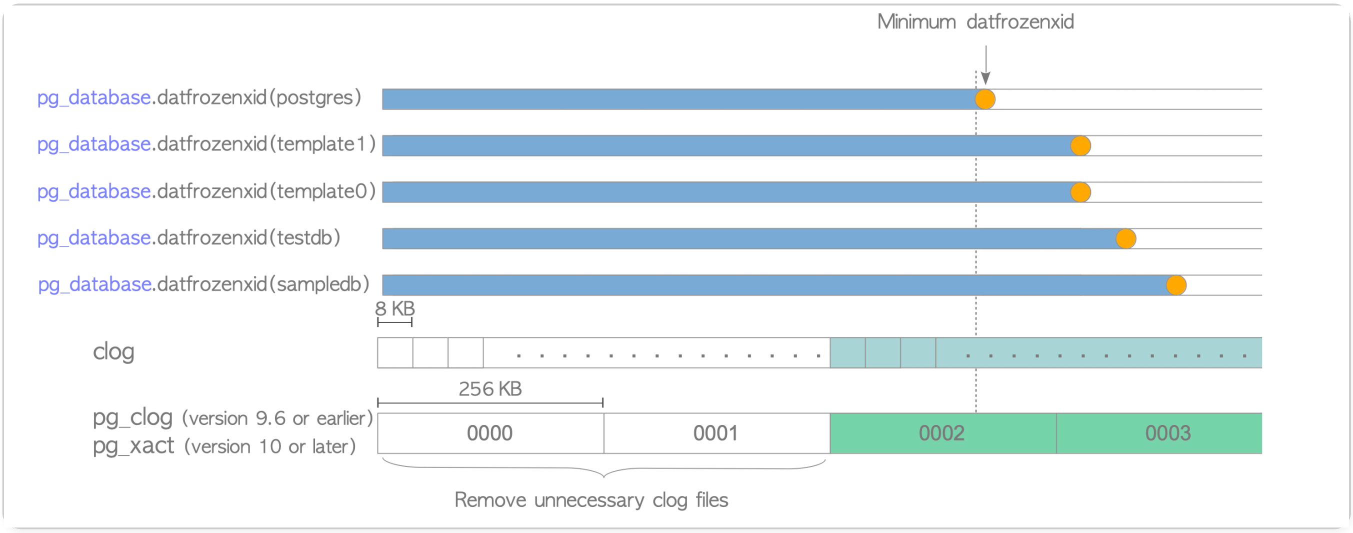


6.4. Removing Unnecessary Clog Files

The clog, described in Section 5.4, stores transaction states. When pg_database.datfrozenxid is updated, PostgreSQL attempts to remove unnecessary clog files. Note that corresponding clog pages are also removed.

Figure 6.7 shows an example. If the minimum pg_database.datfrozenxid is contained in the clog file '0002', the older files ('0000' and '0001') can be removed because all transactions stored in those files can be treated as frozen txids in the whole database cluster.

Fig. 6.7. Removing unnecessary clog files and pages.



Minimum datfrozenxid
pg_database.datfrozenxid(postgres)
pg_database.datfrozenxid(template1)
pg_database.datfrozenxid(template0)
pg_database.datfrozenxid(testdb)
pg_database.datfrozenxid(sampledb)
8 KB
clog
256 KB
pg_clog (version 9.6 or earlier)
pg_xact (version 10 or later)
0000
0001
0002
0003
Remove unnecessary clog files

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 deamon perform vacuum processing concurrently for respective tables, gradually and with minimal impact on database activity.

6.5.1. Conditions for autovacuum to run

The autovacuum process runs for a target table if any of the following conditions are satisfied:

1. The current txid precedes the following expression:

   $$\texttt{relfrozenxid} + \texttt{autovacuum\_freeze\_max\_age},$$

   where **`relfrozenxid`** is the relfrozenxid value of the target table that is defind in the pg_class, and autovacuum_freeze_max_age (the default is 200,000,000) is a configuration parameter.
   If this condition is satisfied, the autovacuum process runs for the target table to perform freeze processing.

2. The number of dead tuples is greater than the following expression:

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

   where autovacuum_vacuum_threshold (the default is 50) and autovacuum_vacuum_scale_factor (the default is 0.2) are configuration parameters, **`reltuples`** is the number of tuples in the target table.
   For example, if the target table has 10,000 tuples and 2,100 dead tuples, the autovacuum process runs for the target table since $2100 > 50 + 0.2 \times 10000$.

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

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

   where autovacuum_vacuum_insert_threshold (the default is 1000) and autovacuum_vacuum_insert_scale_factor (the default is 0.2) are configuration parameters, **`reltuples`** is the number of tuples in the target table.
   For example, if the target table has 10,000 tuples and 3,010 inserted tuples, the autovacuum process runs for the target table since $3010 > 1000 + 0.2 \times 10000$.
   This condition has been added since version 13.

In addition, if the following condition is satisfied for the target table, the autovacuum process will also perform analyze processing.

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

where `mod_since_analyze` is the number of modified tuples (by INSERT, DELETE or UPDATE) since the previous analyze processing, autovacuum_analyze_threshold (the default is 50) and autovacuum_analyze_scale_factor (the default is 0.1) are configuration parameters, `reltuples` is the number of tuples in the target table.

For example, if the target table has 10,000 tuples and 1,100 modified tuples since the previous analyze processing, the autovacuum process will run since $1100 > 50 + 0.1 \times 10000$.

ℹ

The function relation_needs_vacanalyze() determines whether target tables need to be vacuumed or analyzed.

6.5.2. Maintenance tips

As frequently mentioned, table bloat is one of the most annoying things in managing PostgreSQL. Several things can cause that problem, and Autovacuum is one of them.

The autovacuum runs when the number of dead tuples is greater than: 250 for 1,000 relations, 20,050 for 100,000 relations, and 20,000,050 for 100,000,000 relations. It is clear from these examples that the more tuples a table has, the less often autovacuum runs.

A good known tip is to reduce the **autovacuum_vacuum_scale_factor** value. In fact, the default of autovacuum_vacuum_scale_factor (0.2) is too learge for big tables.

PostgreSQL can set an appropriate autovacuum_vacuum_scale_factor in each table using ALTER TABLE command. For example, I show how to set the new value of autovacuum_vacuum_scale_factor for the table pgbench_accounts.

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

If you need that Autovacuum runs for the target tables without depending on the number of their tuples, you can also do it.

For example, assume that you need that Autovacuum processing whenever the number of dead tuples reaches 10,000. In this case, by setting the following storage parameters for the table, the Autovacuum process will perform the vacuum processing each time it reaches 10,000:

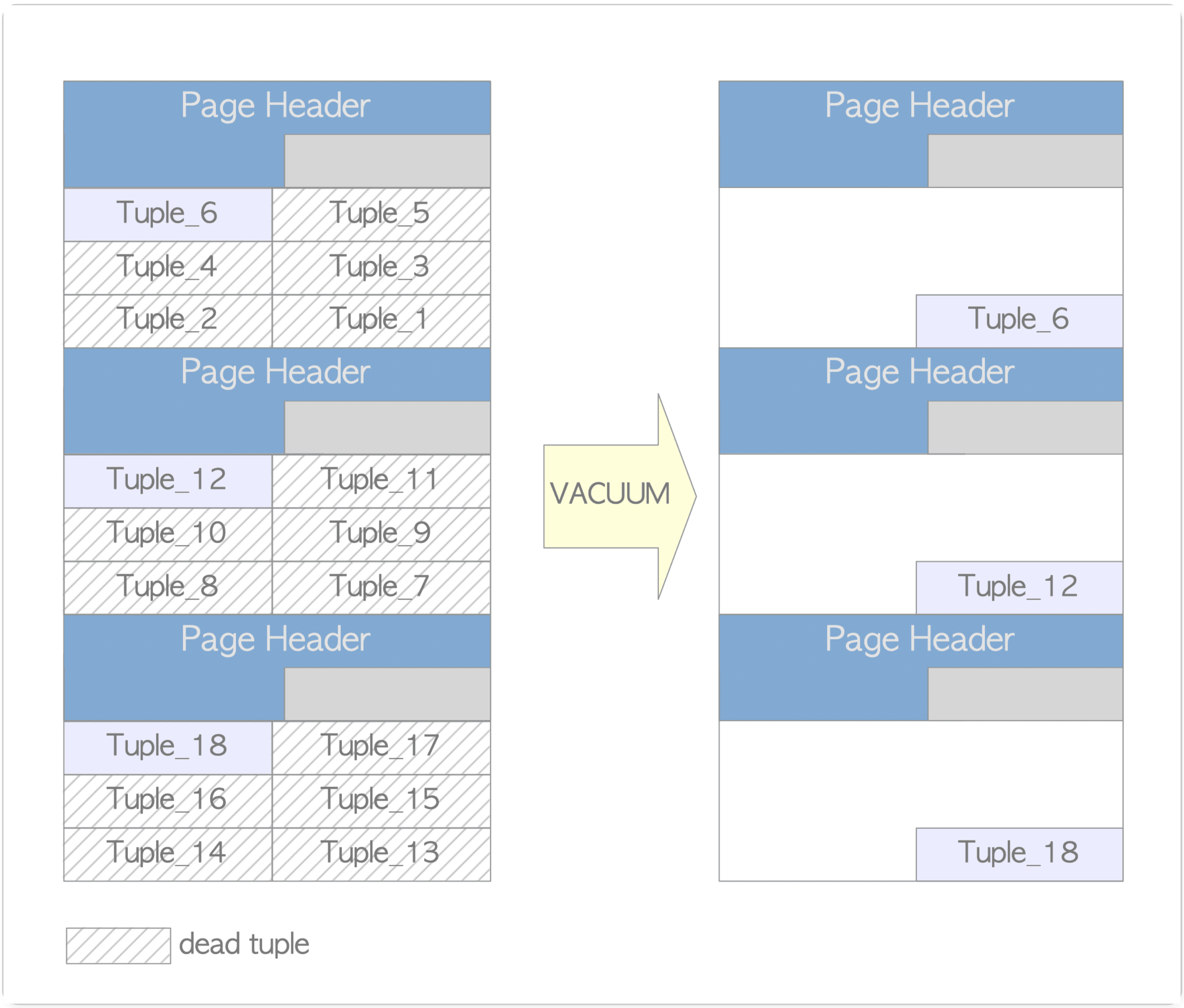
```
postgres=# ALTER TABLE pgbench_accounts SET (autovacuum_vacuum_threshold = 10000);
ALTER TABLE
postgres=# ALTER TABLE pgbench_accounts SET (autovacuum_vacuum_scale_factor = 0.0);
ALTER TABLE
```

6.6. Full VACUUM

Although Concurrent VACUUM is essential for operation, it is not sufficient. For example, it cannot reduce the size of a table even if many dead tuples are removed.

Figure 6.8 shows an extreme example. Suppose that a table consists of three pages, and each page contains six tuples. The following DELETE command is executed to remove tuples, and the VACUUM command is executed to remove dead tuples:

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

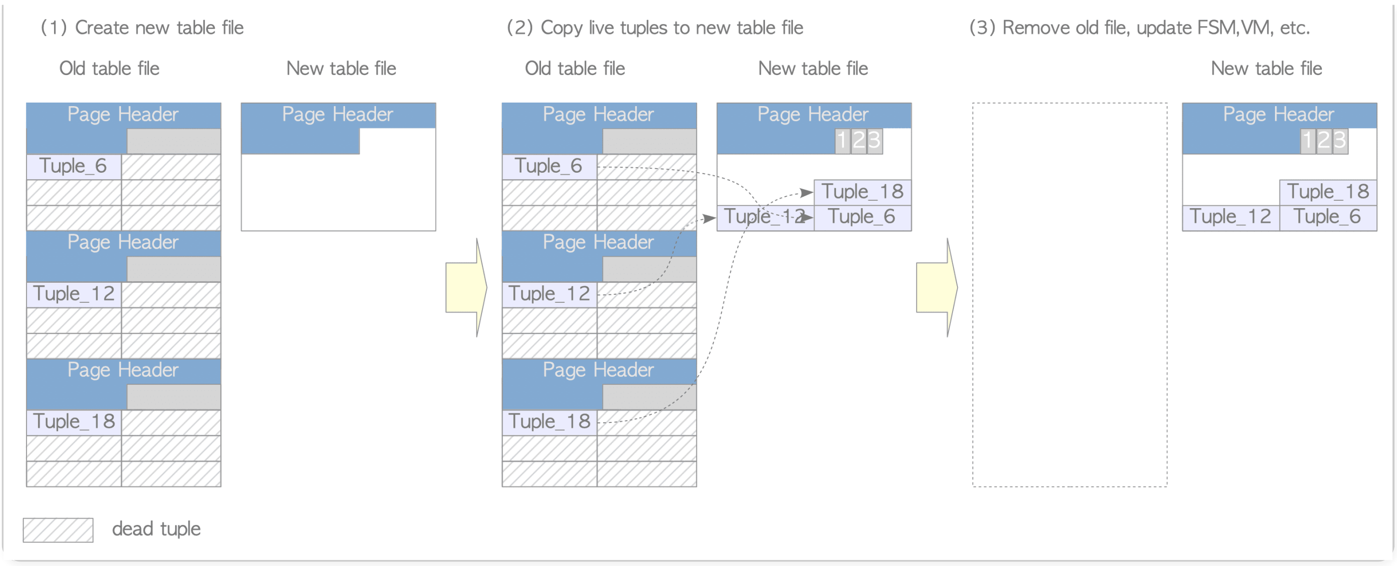


```
testdb=# DELETE FROM tbl WHERE id % 6 != 0;
testdb=# VACUUM tbl;
```

The dead tuples are removed; but the table size is not reduced. This is both a waste of disk space and has a negative impact on database performance. For instance, in the above example, when three tuples in the table are read, three pages must be loaded from disk.

To deal with this situation, PostgreSQL provides the Full VACUUM mode. Figure 6.9 shows an outline of this mode.

Fig. 6.9. Outline of Full VACUUM mode.



[1] Create new table file: Fig. 6.9(1)
When the VACUUM FULL command is executed for a table, PostgreSQL first acquires the AccessExclusiveLock lock for the table and creates a new table file whose size is 8 KB. The AccessExclusiveLock lock prevents other users from accessing the table.

[2] Copy live tuples to the new table: Fig. 6.9(2)
PostgreSQL copies only live tuples within the old table file to the new table.

[3] Remove the old file, rebuild indexes, and update the statistics, FSM, and VM: Fig. 6.9(3)
After copying all live tuples, PostgreSQL removes the old file, rebuilds all associated table indexes, updates both the FSM and VM of this table, and updates associated statistics and system catalogs.

The pseudocode of the Full VACUUM is shown in below:

</> Pseudocode: Full VACUUM

```
(1)  FOR each table
(2)       Acquire AccessExclusiveLock lock for the table
(3)       Create a new table file
(4)       FOR each live tuple in the old table
(5)            Copy the live tuple to the new table file
(6)            Freeze the tuple IF necessary
          END FOR
(7)       Remove the old table file
(8)       Rebuild all indexes
(9)       Update FSM and VM
(10)      Update statistics
          Release AccessExclusiveLock lock
     END FOR
(11)  Remove unnecessary clog files and pages if possible
```

Two points should be considered when using the VACUUM FULL command.

1. Nobody can access(read/write) the table when Full VACUUM is processing.
2. At most twice the disk space of the table is used temporarily; therefore, it is necessary to check the remaining disk capacity when a huge table is processed.

When should I do VACUUM FULL?

There is unfortunately no one-size-fits-all answer to the question of when to execute VACUUM FULL. However, the extension pg_freespacemap can give you some good suggestions.

The following query shows the average freespace ratio of the table you want to know.

```
testdb=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION

testdb=# SELECT count(*) as "number of pages",
       pg_size_pretty(cast(avg(avail) as bigint)) as "Av. freespace size",
       round(100 * avg(avail)/8192 ,2) as "Av. freespace ratio"
       FROM pg_freespace('accounts');
 number of pages | Av. freespace size | Av. freespace ratio
-----------------+--------------------+---------------------
            1640 | 99 bytes           |                1.21
(1 row)
```

As the result above, You can find that there are few free spaces (1.21% free space).

If you delete almost tuples and execute VACUUM command, you can find that almost pages are empty (86.97% free space), but the number of pages remains the same. In other words, the table file has not been compacted.

```
testdb=# DELETE FROM accounts WHERE aid %10 != 0 OR aid < 100;
DELETE 90009

testdb=# VACUUM accounts;
VACUUM

testdb=# SELECT count(*) as "number of pages",
       pg_size_pretty(cast(avg(avail) as bigint)) as "Av. freespace size",
       round(100 * avg(avail)/8192 ,2) as "Av. freespace ratio"
       FROM pg_freespace('accounts');
 number of pages | Av. freespace size | Av. freespace ratio
-----------------+--------------------+---------------------
            1640 | 7124 bytes         |               86.97
(1 row)
```

The following query inspects the freespace ratio of each page of the specified table.

```
testdb=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
                FROM pg_freespace('accounts');
 blkno | avail | freespace ratio
-------+-------+-----------------
     0 |  7904 |           96.00
     1 |  7520 |           91.00
     2 |  7136 |           87.00
     3 |  7136 |           87.00
     4 |  7136 |           87.00
     5 |  7136 |           87.00
....
```

If you run VACUUM FULL in this situation, you will see that the table file has been compacted.

```
testdb=# VACUUM FULL accounts;
VACUUM
testdb=# SELECT count(*) as "number of blocks",
       pg_size_pretty(cast(avg(avail) as bigint)) as "Av. freespace size",
       round(100 * avg(avail)/8192 ,2) as "Av. freespace ratio"
       FROM pg_freespace('accounts');
 number of pages | Av. freespace size | Av. freespace ratio
-----------------+--------------------+---------------------
             164 | 0 bytes            |                0.00
(1 row)
```