



EDB

PG Squeeze

Release 1.6.1

EnterpriseDB Corporation.

Feb 21, 2024

Contents

1	pg_squeeze release notes	3
1.1	Release notes for pg_squeeze version 1.6.1	3
2	Installing pg_squeeze	5
2.1	Installation	5
2.2	Install the package	6
3	Configuring pg_squeeze	7
4	Using pg_squeeze	9
4.1	Registering a table for regular processing	9
4.2	Ad hoc processing for any table	11
4.2.1	Sample execution	11
4.3	Enabling or disabling table processing	11
4.4	Controlling impact on other backends	12
4.5	Running multiple workers per database	12
4.6	Monitoring	13
4.7	Unregistering a table	13
4.8	Concurrency	13
4.9	Disk space requirements	13

PG Squeeze (`pg_squeeze`) is an extension released as open source software under the PostgreSQL License.

`pg_squeeze` removes unused space from a table and optionally sorts tuples according to a particular index (as if a `CLUSTER` command were executed concurrently with regular reads and writes). In fact, `pg_squeeze` tries to replace the `pg_repack` extension. While providing very similar functionality, `pg_squeeze` takes a different approach by:

- Implementing functionality solely on the server side. This approach makes both configuration and use simpler than `pg_repack`, which uses both server- and client-side code. Server-side implementation also allows for smooth implementation of unattended processing using background workers.
- Using recent improvements of the PostgreSQL database server. Most notably, besides the use of background workers, `pg_squeeze` uses logical decode instead of triggers to capture concurrent changes.

For more information about `pg_squeeze`, see:

- *[Installing pg_squeeze](#)*
- *[Configuring pg_squeeze](#)*
- *[Using pg_squeeze](#)*

pg_squeeze release notes

The pg_squeeze documentation describes the latest version of pg_squeeze, including minor releases and patches. These release notes cover what was new in each release. For new functionality introduced in a minor or patch release, there are also indicators in the content about the release that introduced the feature.

Version	Release Date
<i>Release notes for pg_squeeze version 1.6.1</i>	15 Feb 2024

1.1 Release notes for pg_squeeze version 1.6.1

This release of pg_squeeze includes:

Type	Description
Feature	This is the initial release.

CHAPTER 2

Installing pg_squeeze

pg_squeeze is supported on the same platforms as the Postgres distribution you're using. Support for pg_squeeze starts with Postgres 12. For details, see:

- [EDB Postgres Advanced Server Product Compatibility](#)
- [PostgreSQL Product Compatibility](#)
- [EDB Postgres Extended Server Product Compatibility](#)

2.1 Installation

Before you begin the installation process:

- Install Postgres. See:
 - *[Installing EDB Postgres Advanced Server](#)*
 - [Installing PostgreSQL](#)
 - *[Installing EDB Postgres Extended Server](#)*
- Set up the repository.

Setting up the repository is a one-time task. If you've already set up your repository, you don't need to perform this step.

To set up the repository, go to [EDB repositories](#) and follow the instructions provided there.

2.2 Install the package

The syntax for the package install command is:

```
# For SLES, CentOS, RHEL and its derivatives

sudo <package-manager> -y install edb-<postgres><postgres_version>-pg-squeeze1

# For Debian and Ubuntu

sudo <package-manager> -y install edb-<postgres><postgres_version>-pg-squeeze-1
```

Where:

- `<package-manager>` is the package manager used with your operating system:

Package manager	Operating system
dnf	RHEL 8/9 and derivatives
yum	RHEL 7 and derivatives, CentOS 7
zypper	SLES
apt-get	Debian and derivatives

- `<postgres>` is the distribution of Postgres you' re using:

Postgres distribution	Value
PostgreSQL	pg
EDB Postgres Advanced Server	as
EDB Postgres Extended Server	postgrextended

- `<postgres_version>` is the version of Postgres you' re using.

For example, to install `pg_squeeze` for EDB Postgres Advanced Server 15 on a RHEL 8 platform:

```
sudo dnf -y install edb-as15-pg-squeeze1
```

CHAPTER 3

Configuring pg_squeeze

1. Add `pg_squeeze` to the `shared_preload_libraries` parameter:

```
shared_preload_libraries = pg_squeeze
```

Note: If `shared_preload_libraries` has other extensions, then you can add `pg_squeeze` to the list. The order doesn't matter. !!!

1. Add these changes to `postgresql.conf` :

```
wal_level = logical  
max_replication_slots = 1 # ... or add 1 to the current value.
```

1. Restart Postgres.
2. Create the `pg_squeeze` extension in your database:

```
CREATE EXTENSION pg_squeeze;
```

Note: When upgrading a database cluster with `pg_squeeze` installed (either using `pg_dumpall /pg_restore` or `pg_upgrade`), make sure that the new cluster has `pg_squeeze` in `shared_preload_libraries` before you upgrade. Otherwise the upgrade fails.

4.1 Registering a table for regular processing

First, ensure that your table has either a primary key or a unique constraint. This is necessary to process changes resulting from other transactions while `pg_squeeze` is doing its work.

To make the `pg_squeeze` extension aware of the table, you must insert a record into the `squeeze.tables` table. After it is added, table statistics are checked periodically. When the table meets the necessary criteria to be squeezed, a task is added to a queue. The tasks are processed sequentially, in the order they are created.

Here is an example of a simple registration:

```
INSERT INTO squeeze.tables (tabschema, tabname, schedule)
VALUES (public, foo, ({30}, {22}, NULL, NULL, {3, 5}));
```

Additional columns can be specified optionally, for example:

```
INSERT INTO squeeze.tables (
    tabschema,
    tabname,
    schedule,
    free_space_extra,
    vacuum_max_age,
    max_retry
)
VALUES (
    public,
    bar,
    ({30}, {22}, NULL, NULL, {3, 5}),
    30,
    2 hours,
    2
);
```

The following list describes the table metadata:

- **tabschema** — Schema name.
- **tabname** — Table name.
- **schedule** — Specifies when the table should be checked and possibly squeezed. The schedule is described by a value of the following composite data type, which resembles a crontab entry:

```
CREATE TYPE schedule AS (  
    minutes      minute[],  
    hours        hour[],  
    days_of_month dom[],  
    months       month[],  
    days_of_week dow[]  
);
```

In this data type **minutes** (0-59) and **hours** (0-23) determine the time when the check occurs within a day, while **days_of_month** (1-31), **months** (1-12) and **days_of_week** (0-7, where both 0 and 7 stand for Sunday) determine the day of the check.

The check is performed if the **minutes**, **hours**, and **months** all match the current timestamp. NULL means any minute, hour, and month, respectively. Either **days_of_month** or **days_of_week** must match the current timestamp or both must be NULL for the check to take place.

For example, the entries in the sample registration specify to check the table `public.bar` every Wednesday and Friday at 22:30.

- **free_space_extra** — Minimum percentage of extra free space needed to trigger processing of the table. The **extra** adjective refers to the fact that free space derived from **fillfactor** isn't a reason to squeeze the table.

For example, if **fillfactor** equals 60, then at least 40 percent of each page stays free during normal operation. If you want to ensure that 70 percent of free space makes **pg_squeeze** interested in the table, set **free_space_extra** to 30 (that is, 70 percent required to be free minus the 40 percent free due to the **fillfactor**).

The default value of **free_space_extra** is 50.

- **min_size** — Minimum disk space in megabytes that the table must occupy to be eligible for processing. The default value is 8.
- **vacuum_max_age** — Maximum time since the completion of the last **VACUUM** to consider the free space map (FSM) fresh. After this interval has elapsed, the portion of dead tuples might be significant. In this case, you must spend more effort to evaluate the potential effect of **pg_squeeze** beyond checking the FSM. The default value is 1 hour.
- **max_retry** — Maximum number of extra attempts to squeeze a table if the first processing of the corresponding task fails. The typical reason to retry processing is that the table definition changed while the table was being squeezed. If the number of retries is achieved, processing of the table is considered complete. The next task is created at the next scheduled time. The default value of **max_retry** is 0 (that is, don't retry).

Note: The *squeeze.table* is the only table to modify. If you want to change anything else, make sure you understand what you're doing.

4.2 Ad hoc processing for any table

It's possible to squeeze tables manually without registering (that is, without inserting the corresponding record into `squeeze.tables`) and without prior checking of the actual bloat.

Function signature:

```
squeeze.squeeze_table(  
    tabschema name,  
    tabname name,  
    clustering_index name,  
    rel_tablespace name,  
    ind_tablespaces name[]  
)
```

The following list describes the table metadata for ad hoc processing:

- **clustering_index** — Index of the processed table. After processing finishes, tuples of the table are physically sorted by the key of this index.
- **rel_tablespace** — Existing tablespace into which to move the table. NULL means to leave the table where it is.
- **ind_tablespaces** — Two-dimensional array in which each row specifies tablespace mapping of an index. The first and the second columns represent the index name and tablespace name, respectively. All indexes for which no mapping is specified remain in the original tablespace. .. note:: If a tablespace is specified for the table but not for indexes, the table gets moved to that tablespace, but the indexes remain in the original tablespace. In other words, the tablespace of the table isn't the default for indexes.

4.2.1 Sample execution

```
SELECT squeeze.squeeze_table(public, pgbench_accounts);
```

4.3 Enabling or disabling table processing

To enable processing of bloated tables, run this statement as superuser:

```
SELECT squeeze.start_worker();
```

The function starts a background worker (`scheduler worker`) that periodically checks which of the registered tables to check for bloat and creates a task for each. Another worker (`squeeze worker`) is launched whenever a task exists for particular database.

If the scheduler worker is already running for the current database, the function doesn't report any error, and the new worker exits immediately.

If the workers are running for the current database, you can use the following statement to stop them:

```
SELECT squeeze.stop_worker();
```

Note: Only the functions mentioned in this documentation are considered part of the user interface. If you want to call any other function, make sure you understand what you're doing.

If you want the background workers to start during startup of the whole PostgreSQL cluster, add entries like the following to the `postgresql.conf` file:

```
squeeze.worker_autostart = my_database your_database
squeeze.worker_role = postgres
```

The next time you start the cluster, two or more workers (that is, one scheduler worker and one or more squeeze workers) are launched for `my_database` and the same for `your_database`. If you take this approach, any worker doesn't start or stop without doing any work if either:

- The `pg_squeeze` extension doesn't exist in the database.
- The `squeeze.worker_role` parameter specifies a role that doesn't have superuser privileges.

Although there are actually two workers, the functions and configuration variables described here use a singular form of the word `worker`. This is because only one `worker` existed in the previous versions of `pg_squeeze`, which ensured both scheduling and execution of the tasks. This implementation change, then, doesn't force all users to adjust their configuration files during upgrade.

4.4 Controlling impact on other backends

Although the table being squeezed is available for both read and write operations by other transactions most of the time, an exclusive lock is needed to finalize processing. If `pg_squeeze` occasionally seems to block access to tables, consider setting the `squeeze.max_xlock_time` GUC parameter. For example:

```
SET squeeze.max_xlock_time TO 100;
```

This example specifies not to hold the exclusive lock for more than 0.1 second (100 milliseconds). If more time is needed for the final stage, `pg_squeeze` releases the exclusive lock, processes changes committed by other transactions in between, and tries the final stage again. An error is reported if the lock duration is exceeded a few more times. If that happens, either increase the setting or schedule processing of the problematic table for a different time when write activity is lower.

4.5 Running multiple workers per database

If you think that a single squeeze worker doesn't cope with the load, consider setting the `squeeze.workers_per_database` configuration variable to a value higher than 1. Then the `pg_squeeze` extension can process multiple tables simultaneously: one table per squeeze worker.

However, be aware that this setting affects all databases in which you actively use the `pg_squeeze` extension. The total number of all the squeeze workers in the cluster (including the scheduler workers) can't exceed the in-core configuration variable `max_worker_processes`.

4.6 Monitoring

The `squeeze.log` table contains one entry per successfully squeezed table.

The columns `tabschema` and `tabname` identify the processed table. The columns `started` and `finished` report when the processing started and finished. `ins_initial` is the number of tuples inserted into the new table storage during the initial load stage, that is, the number of tuples present in the table before the processing started. On the other hand, `ins`, `upd`, and `del` are the numbers of tuples inserted, updated, and deleted by applications during the table processing. (These concurrent data changes must also be incorporated into the squeezed table. Otherwise they'd get lost.)

The `squeeze.errors` table contains errors that happen during squeezing. A common problem is that someone changes the definition of the table whose processing is in progress (that is, someone adds or removes a column).

The `squeeze.get_active_workers()` function returns a table of squeeze workers that are processing tables in the current database.

The `pid` column contains the system PID of the worker process. The other columns have the same meaning as their counterparts in the `squeeze.log` table. While the `squeeze.log` table shows information only on the completed squeeze operations, the `squeeze.get_active_workers()` function lets you check the progress during processing.

4.7 Unregistering a table

If a particular table is no longer subject to periodical squeezes, delete the corresponding row from the `squeeze.tables` table.

It's also a good practice to unregister a table that you're going to drop, although the background worker does unregister non-existing tables periodically.

4.8 Concurrency

The `pg_squeeze` extension doesn't prevent other transactions from altering a table at certain stages of the processing. If a disruptive command (that is, `ALTER TABLE`, `VACUUM FULL`, `CLUSTER`, or `TRUNCATE`) manages to commit before the squeeze finishes, the `squeeze_table()` function aborts and all changes made to the table are rolled back. The `max_retry` column of the `squeeze.tables` table determines how many times the squeeze worker retries. Changing your schedule might help to avoid these kind of disruptions.

Like `pg_repack`, `pg_squeeze` also changes visibility of rows and thus allows for the MVCC-unsafe behavior described in the first paragraph of [Caveats for MVCC](#).

4.9 Disk space requirements

Performing a full-table squeeze requires twice as much free disk space as the target table and its indexes. For example, if the total size of the tables and indexes to be squeezed is 1GB, you need an additional 2GB of disk space.