PG_Squeeze

PostgreSQL extension that removes unused space from a table and optionally sorts tuples according to particular index (as if CLUSTER command was executed concurrently with regular reads / writes). In fact we try to replace `pg_repack` extension.

While providing very similar functionality, `pg_squeeze` takes a different approach as it:

1. Implements the functionality purely on server side.
2. Utilizes recent improvements of PostgreSQL database server.

While (1) makes both configuration and use simpler (compared to pg_repack which uses both server and client side code), it also allows for rather smooth implementation of unattended processing using background workers.

As for (2), one important difference (besides the use of background workers) is that we use logical decoding instead of triggers to capture concurrent changes.

INSTALL

Install PostgreSQL before proceeding. Make sure to have `pg_config` binary, these are typically included in `-dev` and `-devel` packages.

```
git clone https://github.com/cybertec-postgresql/pg_squeeze.git
cd pg_squeeze
make
make install
```

Set the following in `postgresql.conf`:

```
max_replication_slots = 1 # ... or add 1 to the current value.
shared_preload_libraries = 'pg_squeeze' # ... or add the library to the existing ones.
```

If the PostgreSQL server version is lower than 19, also set this one:

```
wal_level = logical
```

Restart the cluster, and invoke:

```
CREATE EXTENSION pg_squeeze;
```

Note: when upgrading a database cluster with pg_squeeze installed (either using `pg_dumpall`/restore or `pg_upgrade`), make sure that the new cluster has `pg_squeeze` in `shared_preload_libraries` before* you upgrade, otherwise the upgrade will fail.*

Register table for regular processing

First, make sure that your table has an identity index - this is necessary to process changes other transactions might do while `pg_squeeze` is doing its work. If the replica identity of the table is DEFAULT or FULL, primary key constraint provides the identity index. If your table has no primary key, you need to set the identity index explicitly using the ALTER COMMAND ... REPLICA IDENTITY USING INDEX ... command.

To make the `pg_squeeze` extension aware of the table, you need to insert a record into `squeeze.tables` table. Once added, statistics of the table are checked periodically. Whenever the table meets criteria to be "squeezed", a "task" is added to a queue. The tasks are processed sequentially, in the order they were created.

The simplest "registration" looks like:

```
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
);
```

Following is the complete description of table metadata.

- `tabschema` and `tabname` are schema and table name respectively.
- `schedule` column tells 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[]
);
```

Here, `minutes` (0-59) and `hours` (0-23) specify the time of the check 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 `minute`, `hour` and `month` all match the current timestamp, while NULL value means any minute, hour and month respectively. As for `days_of_month` and `days_of_week`, at least one of these needs to match the current timestamp, or both need to be NULL for the check to take place.

For example, in the entries above tell that table `public.bar` should be checked every Wednesday and Friday at 22:30.

- `free_space_extra` is the 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` is not reason to squeeze the table.

  For example, if `fillfactor` is equal to 60, then at least 40 percent of each page should stay 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`).

  Default value of `free_space_extra` is 50.

- `min_size` is the minimum disk space in megabytes the table must occupy to be eligible for processing. The default value is 8.

- `vacuum_max_age` is the maximum time since the completion of the last VACUUM to consider the free space map (FSM) fresh. Once this interval has elapsed, the portion of dead tuples might be significant and so more effort than simply checking the FSM needs to be spent to evaluate the potential effect `pg_squeeze`. The default value is 1 hour.

- `max_retry` is the maximum number of extra attempts to squeeze a table if the first processing of the corresponding task failed. Typical reason to retry the processing is that table definition got 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 as soon as the next scheduled time is reached.

  The default value of `max_retry` is 0 (i.e. do not retry).

- `clustering_index` is an existing index of the processed table. Once the processing is finished, tuples of the table will be physically sorted by the

key of this index.

- `rel_tablespace` is an existing tablespace the table should be moved into. NULL means that the table should stay where it is.
- `ind_tablespaces` is a two-dimensional array in which each row specifies tablespace mapping of an index. The first and the second columns represent index name and tablespace name respectively. All indexes for which no mapping is specified will stay in the original tablespace.

  Regarding tablespaces, one special case is worth to mention: if tablespace is specified for table but not for indexes, the table gets moved to that tablespace but the indexes stay in the original one (i.e. the tablespace of the table is not the default for indexes as one might expect).
- `skip_analyze` indicates that table processing should not be followed by ANALYZE command. The default value is `false`, meaning ANALYZE is performed by default.

`squeeze.table` is the only table user should modify. If you want to change anything else, make sure you perfectly understand what you are doing.

Ad-hoc processing for any table

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

Function signature:

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

Sample execution:

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

Note that the function is not transactional: it only starts a background worker, tells it which table it should process and exits. Rollback of the transaction the function was called in does not revert the changes done by the worker.

Enable / disable 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 should be checked 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 does not report any error but the new worker will exit immediately.

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

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

Only the functions mentioned in this documentation are considered user interface. If you want to call any other one, make sure you perfectly understand what you're doing.

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

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

Next time you start the cluster, two or more workers (i.e. one `scheduler worker` and one or more `squeeze workers`) will be launched for `my_database` and the same for `your_database`. If you take this approach, note that any worker will either reject to start or will stop without doing any work if either:

1. The `pg_squeeze` extension does not exist in the database, or
2. `squeeze.worker_role` parameter specifies role which does not have the superuser privileges.

The functions/configuration variables explained above use singular form of the word `worker` although there are actually two workers. 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 is probably not worth to force all users to adjust their configuration files during upgrade.

Control the impact on other backends

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

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

Tells that the exclusive lock shouldn't be held 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. Error is reported if the lock duration is exceeded a few more times. If that happens, you should either increase the setting or schedule processing of the problematic table to a different daytime, when the write activity is lower.

Running multiple workers per database

If you think that a single squeeze worker does not cope with the load, consider setting the `squeeze.workers_per_database` configuration variable to value higher than 1. Then the `pg_squeeze` extension will be able to process multiple tables at a time - 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") cannot exceed the in-core configuration variable `max_worker_processes`.

Monitoring

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

  The columns `tabschema` and `tabname` identify the processed table. The columns `started` and `finished` tell when the processing started and finished. `ins_initial` is the number of tuples inserted into the new table storage during the "initial load stage", i.e. 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.)

- `squeeze.errors` table contains errors that happened during squeezing. An usual problem reported here is that someone changed definition (e.g. added or removed column) of the table whose processing was just in progress.

- `squeeze.get_active_workers()` function returns a table of squeeze workers which are just 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 only shows information on the completed squeeze operations, the `squeeze.get_active_workers()` function lets you check the progress during the processing.

Unregister table

If particular table should no longer be subject to periodical squeeze, simply delete the corresponding row from `squeeze.tables` table.

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

Upgrade

Make sure to install PostgreSQL and `pg_config`, see install section.

```
make # Compile the newer version.
pg_ctl -D /path/to/cluster stop # Stop the cluster.
make install
pg_ctl -D /path/to/cluster start # Start the cluster.
```

Connect to each database containing `pg_squeeze` and run this command:

```
ALTER EXTENSION pg_squeeze UPDATE;
```

Upgrade from 1.2.x

As there's no straightforward way to migrate the scheduling information (see the notes on the `schedule` column of the `squeeze.tables` table) automatically, and as the `schedule` column must not contain NULL values, the upgrade deletes the contents of the `squeeze.tables` table. Please export the table contents to a file before you perform the upgrade and configure the checks of those tables again as soon as the upgrade is done.

Concurrency

1. The extension does not prevent other transactions from altering table at certain stages of the processing. If a "disruptive command" (i.e. `ALTER TABLE`, `VACUUM FULL`, `CLUSTER` or `TRUNCATE`) manages to commit before the squeeze could finish, the `squeeze_table()` function aborts and all changes done to the table are rolled back. The `max_retry` column of `squeeze.tables` table determines how many times the squeeze worker will retry. Besides that, change of schedule might help you to avoid disruptions.

2. Like `pg_repack`, `pg_squeeze` also changes visibility of rows and thus allows for MVCC-unsafe behavior described in the first paragraph of mvcc-caveats.

Disk Space Requirements

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