**https://www.postgresql.org/docs/current/continuous-archiving.html**

**26.3. Continuous Archiving and Point-in-Time Recovery (PITR)**

[26.3.1. Setting Up WAL Archiving](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-ARCHIVING-WAL)

[26.3.2. Making a Base Backup](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-BASE-BACKUP)

[26.3.3. Making a Base Backup Using the Low Level API](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP)

[26.3.4. Recovering Using a Continuous Archive Backup](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-PITR-RECOVERY)

[26.3.5. Timelines](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-TIMELINES)

[26.3.6. Tips and Examples](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-TIPS)

[26.3.7. Caveats](https://www.postgresql.org/docs/current/continuous-archiving.html#CONTINUOUS-ARCHIVING-CAVEATS)

At all times, PostgreSQL maintains a ***write ahead log* (WAL)** in the **pg\_wal/ subdirectory** of the cluster's data directory. The log records every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by “replaying” the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the file system backup and then replay from the backed-up WAL files to bring the system to a current state. This approach is more complex to administer than either of the previous approaches, but it has some significant benefits:

* We do not need a perfectly consistent file system backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we do not need a file system snapshot capability, just tar or a similar archiving tool.
* Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.
* It is not necessary to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports *point-in-time recovery*: it is possible to restore the database to its state at any time since your base backup was taken.
* If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a *warm standby* system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

**Note**

pg\_dump and pg\_dumpall do not produce file-system-level backups and cannot be used as part of a continuous-archiving solution. Such dumps are *logical* and do not contain enough information to be used by WAL replay.

As with the plain file-system-backup technique, this method can only support restoration of an entire database cluster, not a subset. Also, it requires a lot of archival storage: the base backup might be bulky, and a busy system will generate many megabytes of WAL traffic that have to be archived. Still, it is the preferred backup technique in many situations where high reliability is needed.

To recover successfully using continuous archiving (also called “online backup” by many database vendors), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should set up and test your procedure for archiving WAL files *before* you take your first base backup. Accordingly, we first discuss the mechanics of archiving WAL files.

**26.3.1. Setting Up WAL Archiving**

In an abstract sense, a running PostgreSQL system produces an indefinitely long sequence of WAL records. The system physically divides this sequence into WAL *segment files*, which are normally 16MB apiece (although the segment size can be altered during initdb). The segment files are given numeric names that reflect their position in the abstract WAL sequence. When not using WAL archiving, the system normally creates just a few segment files and then “recycles” them by renaming no-longer-needed segment files to higher segment numbers. It's assumed that segment files whose contents precede the last checkpoint are no longer of interest and can be recycled.

When archiving WAL data, we need to capture the contents of each segment file once it is filled, and save that data somewhere before the segment file is recycled for reuse. Depending on the application and the available hardware, there could be many different ways of “saving the data somewhere”: we could copy the segment files to an NFS-mounted directory on another machine, write them onto a tape drive (ensuring that you have a way of identifying the original name of each file), or batch them together and burn them onto CDs, or something else entirely. To provide the database administrator with flexibility, PostgreSQL tries not to make any assumptions about how the archiving will be done. Instead, **PostgreSQL lets the administrator specify a shell command or an archive library to be executed to copy a completed segment file to wherever it needs to go**. This could be as simple as a shell command that uses cp, or it could invoke a complex C function — it's all up to you.

To enable WAL archiving, set the [**wal\_level**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-WAL-LEVEL) configuration parameter to **replica** or higher, [**archive\_mode**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-ARCHIVE-MODE) to on, specify the shell command to use in the [**archive\_command**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-ARCHIVE-COMMAND) configuration parameter or specify the library to use in the [**archive\_library**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-ARCHIVE-LIBRARY) configuration parameter. **In practice these settings will always be placed in the postgresql.conf file**.

**wal\_level (enum)**wal\_level determines how much information is written to the WAL. The default value is replica, which writes enough data to support WAL archiving and replication, including running read-only queries on a standby server. minimal removes all logging except the information required to recover from a crash or immediate shutdown. Finally, logical adds information necessary to support logical decoding. Each level includes the information logged at all lower levels. This parameter can only be set at server start.

**archive\_mode (enum)**When archive\_mode is enabled, completed WAL segments are sent to archive storage by setting [**archive\_command**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-ARCHIVE-COMMAND) or [**archive\_library**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-ARCHIVE-LIBRARY). In addition to off, to disable, there are two modes: on, and always. During normal operation, there is no difference between the two modes, but when set to always the WAL archiver is enabled also during archive recovery or standby mode. In always mode, all files restored from the archive or streamed with streaming replication will be archived (again). See [**Section 27.2.9**](https://www.postgresql.org/docs/current/warm-standby.html#CONTINUOUS-ARCHIVING-IN-STANDBY) for details.  
archive\_mode is a separate setting from archive\_command and archive\_library so that archive\_command and archive\_library can be changed without leaving archiving mode. This parameter can only be set at server start. archive\_mode cannot be enabled when wal\_level is set to minimal.

In archive\_command, %p is replaced by the path name of the file to archive, while %f is replaced by only the file name. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Use %% if you need to embed an actual % character in the command. The simplest useful command is something like:

archive\_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f' # Unix

archive\_command = 'copy "%p" "C:\\server\\archivedir\\%f"' # Windows

which will copy archivable WAL segments to the directory /mnt/server/archivedir. (This is an example, not a recommendation, and might not work on all platforms.) After the %p and %f parameters have been replaced, the actual command executed might look like this:

test ! -f /mnt/server/archivedir/00000001000000A900000065 && cp pg\_wal/00000001000000A900000065 /mnt/server/archivedir/00000001000000A900000065

A similar command will be generated for each new file to be archived.

The archive command will be executed under the ownership of the same user that the PostgreSQL server is running as. Since the series of WAL files being archived contains effectively everything in your database, you will want to be sure that the archived data is protected from prying eyes; for example, archive into a directory that does not have group or world read access.

It is important that the archive command return zero exit status if and only if it succeeds. Upon getting a zero result, PostgreSQL will assume that the file has been successfully archived, and will remove or recycle it. However, a nonzero status tells PostgreSQL that the file was not archived; it will try again periodically until it succeeds.

Another way to archive is to use a custom archive module as the archive\_library. Since such modules are written in C, creating your own may require considerably more effort than writing a shell command. However, archive modules can be more performant than archiving via shell, and they will have access to many useful server resources. For more information about archive modules, see [**Chapter 51**](https://www.postgresql.org/docs/current/archive-modules.html).

When the archive command is terminated by a signal (other than SIGTERM that is used as part of a server shutdown) or an error by the shell with an exit status greater than 125 (such as command not found), or if the archive function emits an ERROR or FATAL, the archiver process aborts and gets restarted by the postmaster. In such cases, the failure is not reported in [**pg\_stat\_archiver**](https://www.postgresql.org/docs/current/monitoring-stats.html#PG-STAT-ARCHIVER-VIEW).

Archive commands and libraries should generally be designed to refuse to overwrite any pre-existing archive file. This is an important safety feature to preserve the integrity of your archive in case of administrator error (such as sending the output of two different servers to the same archive directory).

It is advisable to test your proposed archive command or library to ensure that it indeed does not overwrite an existing file, *and that it returns nonzero status or false, respectively, in this case*. The example command above for Unix ensures this by including a separate test step. On some Unix platforms, cp has switches such as -i that can be used to do the same thing less verbosely, but you should not rely on these without verifying that the right exit status is returned. (In particular, GNU cp will return status zero when -i is used and the target file already exists, which is *not* the desired behavior.)

While designing your archiving setup, consider what will happen if the archive command or library fails repeatedly because some aspect requires operator intervention or the archive runs out of space. For example, this could occur if you write to tape without an autochanger; when the tape fills, nothing further can be archived until the tape is swapped. You should ensure that any error condition or request to a human operator is reported appropriately so that the situation can be resolved reasonably quickly. The pg\_wal/ directory will continue to fill with WAL segment files until the situation is resolved. (If the file system containing pg\_wal/ fills up, PostgreSQL will do a PANIC shutdown. No committed transactions will be lost, but the database will remain offline until you free some space.)

The speed of the archive command or library is unimportant as long as it can keep up with the average rate at which your server generates WAL data. Normal operation continues even if the archiving process falls a little behind. If archiving falls significantly behind, this will increase the amount of data that would be lost in the event of a disaster. It will also mean that the pg\_wal/ directory will contain large numbers of not-yet-archived segment files, which could eventually exceed available disk space. You are advised to monitor the archiving process to ensure that it is working as you intend.

In writing your archive command or library, you should assume that the file names to be archived can be up to 64 characters long and can contain any combination of ASCII letters, digits, and dots. It is not necessary to preserve the original relative path (%p) but it is necessary to preserve the file name (%f).

Note that although WAL archiving will allow you to restore any modifications made to the data in your PostgreSQL database, it will not restore changes made to configuration files (that is, postgresql.conf, pg\_hba.conf and pg\_ident.conf), since those are edited manually rather than through SQL operations. You might wish to keep the configuration files in a location that will be backed up by your regular file system backup procedures. See [**Section 20.2**](https://www.postgresql.org/docs/current/runtime-config-file-locations.html) for how to relocate the configuration files.

The archive command or function is only invoked on completed WAL segments. Hence, if your server generates only little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set [**archive\_timeout**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-ARCHIVE-TIMEOUT) to force the server to switch to a new WAL segment file at least that often. Note that archived files that are archived early due to a forced switch are still the same length as completely full files. It is therefore unwise to set a very short archive\_timeout — it will bloat your archive storage. archive\_timeout settings of a minute or so are usually reasonable.

Also, you can force a segment switch manually with pg\_switch\_wal if you want to ensure that a just-finished transaction is archived as soon as possible. Other utility functions related to WAL management are listed in [**Table 9.89**](https://www.postgresql.org/docs/current/functions-admin.html#FUNCTIONS-ADMIN-BACKUP-TABLE).

When wal\_level is minimal some SQL commands are optimized to avoid WAL logging, as described in [**Section 14.4.7**](https://www.postgresql.org/docs/current/populate.html#POPULATE-PITR). If archiving or streaming replication were turned on during execution of one of these statements, WAL would not contain enough information for archive recovery. (Crash recovery is unaffected.) For this reason, wal\_level can only be changed at server start. However, archive\_command and archive\_library can be changed with a configuration file reload. If you are archiving via shell and wish to temporarily stop archiving, one way to do it is to set archive\_command to the empty string (''). This will cause WAL files to accumulate in pg\_wal/ until a working archive\_command is re-established.

**26.3.2. Making a Base Backup**

The easiest way to perform a base backup is to use the [**pg\_basebackup**](https://www.postgresql.org/docs/current/app-pgbasebackup.html) tool. It can create a base backup either as regular files or as a tar archive. If more flexibility than [**pg\_basebackup**](https://www.postgresql.org/docs/current/app-pgbasebackup.html) can provide is required, you can also make a base backup using the low level API (see [**Section 26.3.3**](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP)).

It is not necessary to be concerned about the amount of time it takes to make a base backup. However, if you normally run the server with full\_page\_writes disabled, you might notice a drop in performance while the backup runs since full\_page\_writes is effectively forced on during backup mode.

To make use of the backup, you will need to keep all the WAL segment files generated during and after the file system backup. To aid you in doing this, the base backup process creates a *backup history file* that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need for the file system backup. For example, if the starting WAL file is 0000000100001234000055CD the backup history file will be named something like 0000000100001234000055CD.007C9330.backup. (The second part of the file name stands for an exact position within the WAL file, and can ordinarily be ignored.) Once you have safely archived the file system backup and the WAL segment files used during the backup (as specified in the backup history file), all archived WAL segments with names numerically less are no longer needed to recover the file system backup and can be deleted. However, you should consider keeping several backup sets to be absolutely certain that you can recover your data.

The backup history file is just a small text file. It contains the label string you gave to [**pg\_basebackup**](https://www.postgresql.org/docs/current/app-pgbasebackup.html), as well as the starting and ending times and WAL segments of the backup. If you used the label to identify the associated dump file, then the archived history file is enough to tell you which dump file to restore.

Since you have to keep around all the archived WAL files back to your last base backup, the interval between base backups should usually be chosen based on how much storage you want to expend on archived WAL files. You should also consider how long you are prepared to spend recovering, if recovery should be necessary — the system will have to replay all those WAL segments, and that could take awhile if it has been a long time since the last base backup.

**26.3.3. Making a Base Backup Using the Low Level API**

The procedure for making a base backup using the low level APIs contains a few more steps than the [**pg\_basebackup**](https://www.postgresql.org/docs/current/app-pgbasebackup.html) method, but is relatively simple. It is very important that these steps are executed in sequence, and that the success of a step is verified before proceeding to the next step.

Multiple backups are able to be run concurrently (both those started using this backup API and those started using [**pg\_basebackup**](https://www.postgresql.org/docs/current/app-pgbasebackup.html)).

1. Ensure that WAL archiving is enabled and working.
2. Connect to the server (it does not matter which database) as a user with rights to run pg\_backup\_start (superuser, or a user who has been granted EXECUTE on the function) and issue the command:
3. SELECT pg\_backup\_start(label => 'label', fast => false);

where label is any string you want to use to uniquely identify this backup operation. The connection calling pg\_backup\_start must be maintained until the end of the backup, or the backup will be automatically aborted.

Online backups are always started at the beginning of a checkpoint. By default, pg\_backup\_start will wait for the next regularly scheduled checkpoint to complete, which may take a long time (see the configuration parameters [**checkpoint\_timeout**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-CHECKPOINT-TIMEOUT) and [**checkpoint\_completion\_target**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-CHECKPOINT-COMPLETION-TARGET)). This is usually preferable as it minimizes the impact on the running system. If you want to start the backup as soon as possible, pass true as the second parameter to pg\_backup\_start and it will request an immediate checkpoint, which will finish as fast as possible using as much I/O as possible.

1. Perform the backup, using any convenient file-system-backup tool such as tar or cpio (not pg\_dump or pg\_dumpall). It is neither necessary nor desirable to stop normal operation of the database while you do this. See [**Section 26.3.3.1**](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-DATA) for things to consider during this backup.
2. In the same connection as before, issue the command:
3. SELECT \* FROM pg\_backup\_stop(wait\_for\_archive => true);

This terminates backup mode. On a primary, it also performs an automatic switch to the next WAL segment. On a standby, it is not possible to automatically switch WAL segments, so you may wish to run pg\_switch\_wal on the primary to perform a manual switch. The reason for the switch is to arrange for the last WAL segment file written during the backup interval to be ready to archive.

pg\_backup\_stop will return one row with three values. The second of these fields should be written to a file named backup\_label in the root directory of the backup. The third field should be written to a file named tablespace\_map unless the field is empty. These files are vital to the backup working and must be written byte for byte without modification, which may require opening the file in binary mode.

1. Once the WAL segment files active during the backup are archived, you are done. The file identified by pg\_backup\_stop's first return value is the last segment that is required to form a complete set of backup files. On a primary, if archive\_mode is enabled and the wait\_for\_archive parameter is true, pg\_backup\_stop does not return until the last segment has been archived. On a standby, archive\_mode must be always in order for pg\_backup\_stop to wait. Archiving of these files happens automatically since you have already configured archive\_command or archive\_library. In most cases this happens quickly, but you are advised to monitor your archive system to ensure there are no delays. If the archive process has fallen behind because of failures of the archive command or library, it will keep retrying until the archive succeeds and the backup is complete. If you wish to place a time limit on the execution of pg\_backup\_stop, set an appropriate statement\_timeout value, but make note that if pg\_backup\_stop terminates because of this your backup may not be valid.

If the backup process monitors and ensures that all WAL segment files required for the backup are successfully archived then the wait\_for\_archive parameter (which defaults to true) can be set to false to have pg\_backup\_stop return as soon as the stop backup record is written to the WAL. By default, pg\_backup\_stop will wait until all WAL has been archived, which can take some time. This option must be used with caution: if WAL archiving is not monitored correctly then the backup might not include all of the WAL files and will therefore be incomplete and not able to be restored.

**26.3.3.1. Backing Up The Data Directory**

Some file system backup tools emit warnings or errors if the files they are trying to copy change while the copy proceeds. When taking a base backup of an active database, this situation is normal and not an error. However, you need to ensure that you can distinguish complaints of this sort from real errors. For example, some versions of rsync return a separate exit code for “vanished source files”, and you can write a driver script to accept this exit code as a non-error case. Also, some versions of GNU tar return an error code indistinguishable from a fatal error if a file was truncated while tar was copying it. Fortunately, GNU tar versions 1.16 and later exit with 1 if a file was changed during the backup, and 2 for other errors. With GNU tar version 1.23 and later, you can use the warning options --warning=no-file-changed --warning=no-file-removed to hide the related warning messages.

Be certain that your backup includes all of the files under the database cluster directory (e.g., /usr/local/pgsql/data). If you are using tablespaces that do not reside underneath this directory, be careful to include them as well (and be sure that your backup archives symbolic links as links, otherwise the restore will corrupt your tablespaces).

You should, however, omit from the backup the files within the cluster's pg\_wal/ subdirectory. This slight adjustment is worthwhile because it reduces the risk of mistakes when restoring. This is easy to arrange if pg\_wal/ is a symbolic link pointing to someplace outside the cluster directory, which is a common setup anyway for performance reasons. You might also want to exclude postmaster.pid and postmaster.opts, which record information about the running postmaster, not about the postmaster which will eventually use this backup. (These files can confuse pg\_ctl.)

It is often a good idea to also omit from the backup the files within the cluster's pg\_replslot/ directory, so that replication slots that exist on the primary do not become part of the backup. Otherwise, the subsequent use of the backup to create a standby may result in indefinite retention of WAL files on the standby, and possibly bloat on the primary if hot standby feedback is enabled, because the clients that are using those replication slots will still be connecting to and updating the slots on the primary, not the standby. Even if the backup is only intended for use in creating a new primary, copying the replication slots isn't expected to be particularly useful, since the contents of those slots will likely be badly out of date by the time the new primary comes on line.

The contents of the directories pg\_dynshmem/, pg\_notify/, pg\_serial/, pg\_snapshots/, pg\_stat\_tmp/, and pg\_subtrans/ (but not the directories themselves) can be omitted from the backup as they will be initialized on postmaster startup.

Any file or directory beginning with pgsql\_tmp can be omitted from the backup. These files are removed on postmaster start and the directories will be recreated as needed.

pg\_internal.init files can be omitted from the backup whenever a file of that name is found. These files contain relation cache data that is always rebuilt when recovering.

The backup label file includes the label string you gave to pg\_backup\_start, as well as the time at which pg\_backup\_start was run, and the name of the starting WAL file. In case of confusion it is therefore possible to look inside a backup file and determine exactly which backup session the dump file came from. The tablespace map file includes the symbolic link names as they exist in the directory pg\_tblspc/ and the full path of each symbolic link. These files are not merely for your information; their presence and contents are critical to the proper operation of the system's recovery process.

It is also possible to make a backup while the server is stopped. In this case, you obviously cannot use pg\_backup\_start or pg\_backup\_stop, and you will therefore be left to your own devices to keep track of which backup is which and how far back the associated WAL files go. It is generally better to follow the continuous archiving procedure above.

**26.3.4. Recovering Using a Continuous Archive Backup**

Okay, the worst has happened and you need to recover from your backup. Here is the procedure:

1. Stop the server, if it's running.
2. If you have the space to do so, copy the whole cluster data directory and any tablespaces to a temporary location in case you need them later. Note that this precaution will require that you have enough free space on your system to hold two copies of your existing database. If you do not have enough space, you should at least save the contents of the cluster's pg\_wal subdirectory, as it might contain logs which were not archived before the system went down.
3. Remove all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.
4. Restore the database files from your file system backup. Be sure that they are restored with the right ownership (the database system user, not root!) and with the right permissions. If you are using tablespaces, you should verify that the symbolic links in pg\_tblspc/ were correctly restored.
5. Remove any files present in pg\_wal/; these came from the file system backup and are therefore probably obsolete rather than current. If you didn't archive pg\_wal/ at all, then recreate it with proper permissions, being careful to ensure that you re-establish it as a symbolic link if you had it set up that way before.
6. If you have unarchived WAL segment files that you saved in step 2, copy them into pg\_wal/. (It is best to copy them, not move them, so you still have the unmodified files if a problem occurs and you have to start over.)
7. Set recovery configuration settings in postgresql.conf (see [**Section 20.5.5**](https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-ARCHIVE-RECOVERY)) and create a file recovery.signal in the cluster data directory. You might also want to temporarily modify pg\_hba.conf to prevent ordinary users from connecting until you are sure the recovery was successful.
8. Start the server. The server will go into recovery mode and proceed to read through the archived WAL files it needs. Should the recovery be terminated because of an external error, the server can simply be restarted and it will continue recovery. Upon completion of the recovery process, the server will remove recovery.signal (to prevent accidentally re-entering recovery mode later) and then commence normal database operations.
9. Inspect the contents of the database to ensure you have recovered to the desired state. If not, return to step 1. If all is well, allow your users to connect by restoring pg\_hba.conf to normal.

The key part of all this is to set up a recovery configuration that describes how you want to recover and how far the recovery should run. The one thing that you absolutely must specify is the restore\_command, which tells PostgreSQL how to retrieve archived WAL file segments. Like the archive\_command, this is a shell command string. It can contain %f, which is replaced by the name of the desired log file, and %p, which is replaced by the path name to copy the log file to. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Write %% if you need to embed an actual % character in the command. The simplest useful command is something like:

restore\_command = 'cp /mnt/server/archivedir/%f %p'

which will copy previously archived WAL segments from the directory /mnt/server/archivedir. Of course, you can use something much more complicated, perhaps even a shell script that requests the operator to mount an appropriate tape.

It is important that the command return nonzero exit status on failure. The command *will* be called requesting files that are not present in the archive; it must return nonzero when so asked. This is not an error condition. An exception is that if the command was terminated by a signal (other than SIGTERM, which is used as part of a database server shutdown) or an error by the shell (such as command not found), then recovery will abort and the server will not start up.

Not all of the requested files will be WAL segment files; you should also expect requests for files with a suffix of .history. Also be aware that the base name of the %p path will be different from %f; do not expect them to be interchangeable.

WAL segments that cannot be found in the archive will be sought in pg\_wal/; this allows use of recent un-archived segments. However, segments that are available from the archive will be used in preference to files in pg\_wal/.

Normally, recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time (or as close as possible given the available WAL segments). Therefore, a normal recovery will end with a “file not found” message, the exact text of the error message depending upon your choice of restore\_command. You may also see an error message at the start of recovery for a file named something like 00000001.history. This is also normal and does not indicate a problem in simple recovery situations; see [**Section 26.3.5**](https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-TIMELINES) for discussion.

If you want to recover to some previous point in time (say, right before the junior DBA dropped your main transaction table), just specify the required [**stopping point**](https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET). You can specify the stop point, known as the “recovery target”, either by date/time, named restore point or by completion of a specific transaction ID. As of this writing only the date/time and named restore point options are very usable, since there are no tools to help you identify with any accuracy which transaction ID to use.

**Note**

The stop point must be after the ending time of the base backup, i.e., the end time of pg\_backup\_stop. You cannot use a base backup to recover to a time when that backup was in progress. (To recover to such a time, you must go back to your previous base backup and roll forward from there.)

If recovery finds corrupted WAL data, recovery will halt at that point and the server will not start. In such a case the recovery process could be re-run from the beginning, specifying a “recovery target” before the point of corruption so that recovery can complete normally. If recovery fails for an external reason, such as a system crash or if the WAL archive has become inaccessible, then the recovery can simply be restarted and it will restart almost from where it failed. Recovery restart works much like checkpointing in normal operation: the server periodically forces all its state to disk, and then updates the pg\_control file to indicate that the already-processed WAL data need not be scanned again.

**26.3.5. Timelines**

The ability to restore the database to a previous point in time creates some complexities that are akin to science-fiction stories about time travel and parallel universes. For example, in the original history of the database, suppose you dropped a critical table at 5:15PM on Tuesday evening, but didn't realize your mistake until Wednesday noon. Unfazed, you get out your backup, restore to the point-in-time 5:14PM Tuesday evening, and are up and running. In *this* history of the database universe, you never dropped the table. But suppose you later realize this wasn't such a great idea, and would like to return to sometime Wednesday morning in the original history. You won't be able to if, while your database was up-and-running, it overwrote some of the WAL segment files that led up to the time you now wish you could get back to. Thus, to avoid this, you need to distinguish the series of WAL records generated after you've done a point-in-time recovery from those that were generated in the original database history.

To deal with this problem, PostgreSQL has a notion of *timelines*. Whenever an archive recovery completes, a new timeline is created to identify the series of WAL records generated after that recovery. The timeline ID number is part of WAL segment file names so a new timeline does not overwrite the WAL data generated by previous timelines. It is in fact possible to archive many different timelines. While that might seem like a useless feature, it's often a lifesaver. Consider the situation where you aren't quite sure what point-in-time to recover to, and so have to do several point-in-time recoveries by trial and error until you find the best place to branch off from the old history. Without timelines this process would soon generate an unmanageable mess. With timelines, you can recover to *any* prior state, including states in timeline branches that you abandoned earlier.

Every time a new timeline is created, PostgreSQL creates a “timeline history” file that shows which timeline it branched off from and when. These history files are necessary to allow the system to pick the right WAL segment files when recovering from an archive that contains multiple timelines. Therefore, they are archived into the WAL archive area just like WAL segment files. The history files are just small text files, so it's cheap and appropriate to keep them around indefinitely (unlike the segment files which are large). You can, if you like, add comments to a history file to record your own notes about how and why this particular timeline was created. Such comments will be especially valuable when you have a thicket of different timelines as a result of experimentation.

The default behavior of recovery is to recover to the latest timeline found in the archive. If you wish to recover to the timeline that was current when the base backup was taken or into a specific child timeline (that is, you want to return to some state that was itself generated after a recovery attempt), you need to specify current or the target timeline ID in [**recovery\_target\_timeline**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-RECOVERY-TARGET-TIMELINE). You cannot recover into timelines that branched off earlier than the base backup.

**26.3.6. Tips and Examples**

Some tips for configuring continuous archiving are given here.

**26.3.6.1. Standalone Hot Backups**

It is possible to use PostgreSQL's backup facilities to produce standalone hot backups. These are backups that cannot be used for point-in-time recovery, yet are typically much faster to backup and restore than pg\_dump dumps. (They are also much larger than pg\_dump dumps, so in some cases the speed advantage might be negated.)

As with base backups, the easiest way to produce a standalone hot backup is to use the [**pg\_basebackup**](https://www.postgresql.org/docs/current/app-pgbasebackup.html) tool. If you include the -X parameter when calling it, all the write-ahead log required to use the backup will be included in the backup automatically, and no special action is required to restore the backup.

**26.3.6.2. Compressed Archive Logs**

If archive storage size is a concern, you can use gzip to compress the archive files:

archive\_command = 'gzip < %p > /mnt/server/archivedir/%f.gz'

You will then need to use gunzip during recovery:

restore\_command = 'gunzip < /mnt/server/archivedir/%f.gz > %p'

**26.3.6.3.** Archive\_command **Scripts**

Many people choose to use scripts to define their archive\_command, so that their postgresql.conf entry looks very simple:

archive\_command = 'local\_backup\_script.sh "%p" "%f"'

Using a separate script file is advisable any time you want to use more than a single command in the archiving process. This allows all complexity to be managed within the script, which can be written in a popular scripting language such as bash or perl.

Examples of requirements that might be solved within a script include:

* Copying data to secure off-site data storage
* Batching WAL files so that they are transferred every three hours, rather than one at a time
* Interfacing with other backup and recovery software
* Interfacing with monitoring software to report errors

**Tip**

When using an archive\_command script, it's desirable to enable [**logging\_collector**](https://www.postgresql.org/docs/current/runtime-config-logging.html#GUC-LOGGING-COLLECTOR). Any messages written to stderr from the script will then appear in the database server log, allowing complex configurations to be diagnosed easily if they fail.

**26.3.7. Caveats**

At this writing, there are several limitations of the continuous archiving technique. These will probably be fixed in future releases:

* If a [CREATE DATABASE](https://www.postgresql.org/docs/current/sql-createdatabase.html) command is executed while a base backup is being taken, and then the template database that the CREATE DATABASE copied is modified while the base backup is still in progress, it is possible that recovery will cause those modifications to be propagated into the created database as well. This is of course undesirable. To avoid this risk, it is best not to modify any template databases while taking a base backup.
* [CREATE TABLESPACE](https://www.postgresql.org/docs/current/sql-createtablespace.html) commands are WAL-logged with the literal absolute path, and will therefore be replayed as tablespace creations with the same absolute path. This might be undesirable if the log is being replayed on a different machine. It can be dangerous even if the log is being replayed on the same machine, but into a new data directory: the replay will still overwrite the contents of the original tablespace. To avoid potential gotchas of this sort, the best practice is to take a new base backup after creating or dropping tablespaces.

It should also be noted that the default WAL format is fairly bulky since it includes many disk page snapshots. These page snapshots are designed to support crash recovery, since we might need to fix partially-written disk pages. Depending on your system hardware and software, the risk of partial writes might be small enough to ignore, in which case you can significantly reduce the total volume of archived logs by turning off page snapshots using the [**full\_page\_writes**](https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-FULL-PAGE-WRITES) parameter. (Read the notes and warnings in [**Chapter 30**](https://www.postgresql.org/docs/current/wal.html) before you do so.) Turning off page snapshots does not prevent use of the logs for PITR operations. An area for future development is to compress archived WAL data by removing unnecessary page copies even when full\_page\_writes is on. In the meantime, administrators might wish to reduce the number of page snapshots included in WAL by increasing the checkpoint interval parameters as much as feasible.

|  |  |  |
| --- | --- | --- |
| [**Prev**](https://www.postgresql.org/docs/current/backup-file.html) | [**Up**](https://www.postgresql.org/docs/current/backup.html) | [**Next**](https://www.postgresql.org/docs/current/high-availability.html) |
| 26.2. File System Level Backup | [**Home**](https://www.postgresql.org/docs/current/index.html) | Chapter 27. High Availability, Load Balancing, and Replication |

**Submit correction**

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use [**this form**](https://www.postgresql.org/account/comments/new/15/continuous-archiving.html/) to report a documentation issue.

# [**Query for all the Postgres configuration parameters‘ current values?**](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values)

[Ask Question](https://dba.stackexchange.com/questions/ask)

Asked 4 years, 7 months ago

Modified [3 years, 11 months ago](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values?lastactivity)

Viewed 56k times

Report this ad

17

You can [set various configuration parameters](https://www.postgresql.org/docs/current/static/config-setting.html) for Postgres by either editing the **postgresql.conf** file manually or by calling [ALTER SYSTEM](https://www.postgresql.org/docs/current/static/sql-altersystem.html) commands. These are two avenues for writing the settings, but how about reading?

➥ Is there a way to query for all the current settings?

I know the settings in the client authentication configuration file pg\_hba.conf can be read with the [pg\_hba\_file\_rules](https://www.postgresql.org/docs/current/static/view-pg-hba-file-rules.html) view, like this: table pg\_hba\_file\_rules ;. Is there something similar for postgresql.conf?

[Basil Bourque](https://dba.stackexchange.com/users/19079/basil-bourque)

**10.1k**1818 gold badges5353 silver badges9090 bronze badges

* 3

select \* from pg\_settings?

– [a\_horse\_with\_no\_name](https://dba.stackexchange.com/users/1822/a-horse-with-no-name)

[Oct 24, 2018 at 19:29](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment433557_220933)

* @a\_horse\_with\_no\_name Thanks. I posted an Answer based on your comment.

– [Basil Bourque](https://dba.stackexchange.com/users/19079/basil-bourque)

[Oct 24, 2018 at 19:50](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment433561_220933)

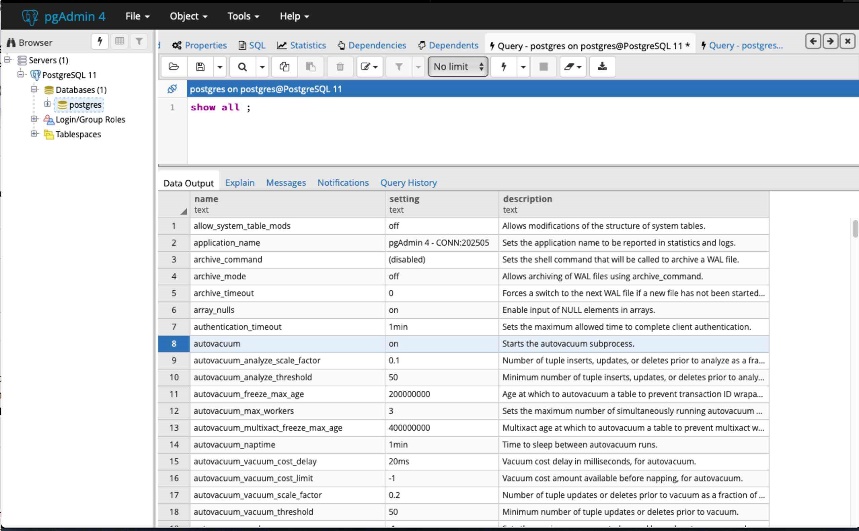
[Add a comment](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values)

## 1 Answer

## SHOW ALL

The [SHOW ALL](https://www.postgresql.org/docs/current/static/sql-show.html) command displays the current setting of run-time parameters in 3 columns.

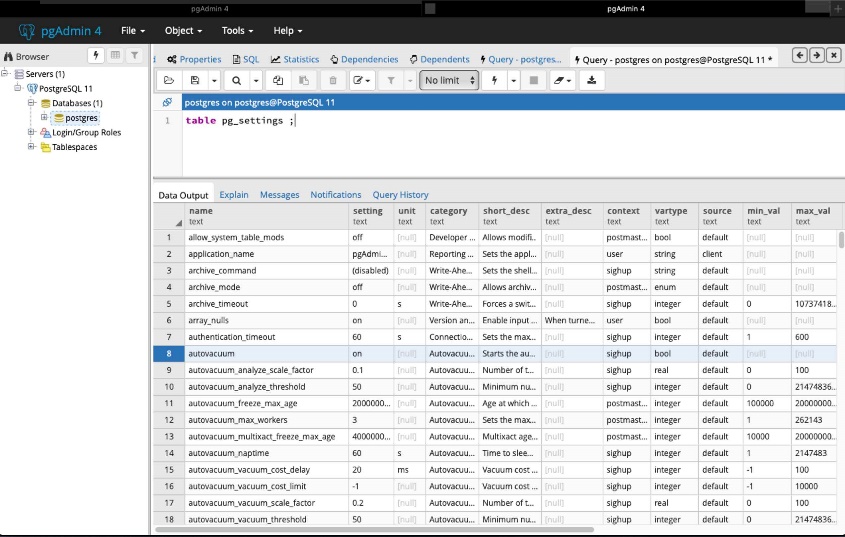
SHOW ALL ;

[](https://i.stack.imgur.com/uCDO1.jpg)

# pg\_settings

The [pg\_settings](https://www.postgresql.org/docs/current/static/view-pg-settings.html) view shows the same items as [SHOW ALL](https://www.postgresql.org/docs/current/static/sql-show.html) but with additional details, across 17 columns versus 3 columns.

TABLE pg\_settings ;

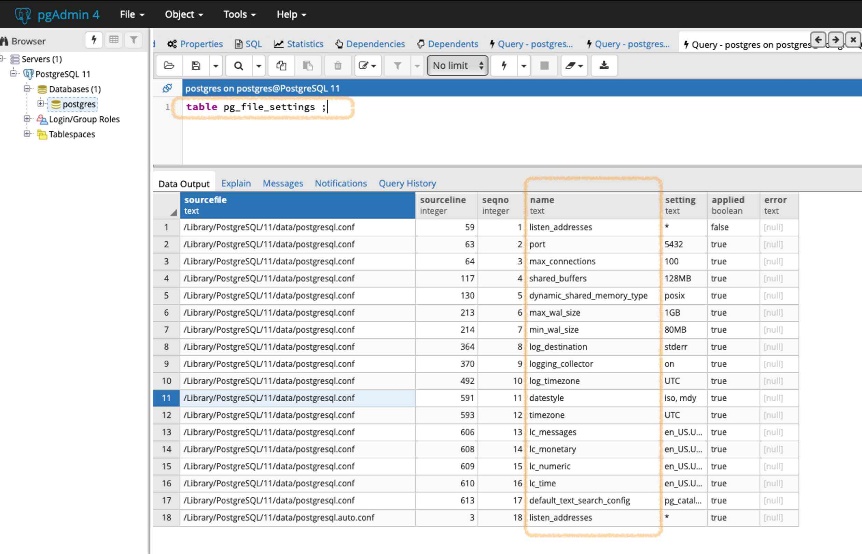
[](https://i.stack.imgur.com/38dVM.jpg)

# pg\_file\_settings

To read what is stored in the postgresql.conf file itself, use the view [pg\_file\_settings](https://www.postgresql.org/docs/current/static/view-pg-file-settings.html).

If freshly written, this file will hold values that may not yet be in effect within the Postgres server. After writing, the settings must be loaded in one of various ways including a server restart.

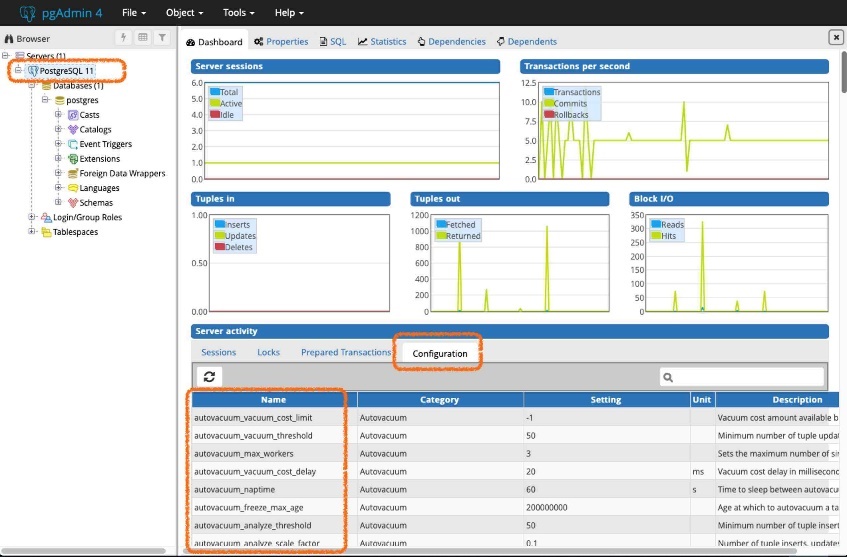
TABLE pg\_file\_settings ;

[](https://i.stack.imgur.com/ooceY.jpg)

# Admin tool

Your databased-administration tool may display these settings.

For example, in [pgAdmin 4](https://www.pgadmin.org/), choose the cluster name (the Postgres installation) in the navigation panel, click the Dashboard tab, and in a list bottom panel titled Server activity, click the Configuration tab to see a list of your settings.

[](https://i.stack.imgur.com/qEe8Z.jpg)

# Writing

By the way… For info about writing these settings with ALTER SYSTEM commands, see [my Answer](https://dba.stackexchange.com/a/220836/19079) on another Question, [How to edit *postgresql.conf* with pgAdmin 4?](https://dba.stackexchange.com/q/220700/19079).

[Share](https://dba.stackexchange.com/a/220937)

* Is TABLE pg\_file\_settings; equivalent to SELECT \* FROM pg\_file\_settings;?

– [Rafs](https://dba.stackexchange.com/users/93601/rafs)

[May 30, 2022 at 16:00](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment609245_220937)

* @RTD Yes. The first is a convenient shortcut specific to Postgres, the second is standard SQL.

– [Basil Bourque](https://dba.stackexchange.com/users/19079/basil-bourque)

[May 30, 2022 at 16:10](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment609247_220937)

* Cool, thanks. If you have a piece of documentation for this shortcut, I'd be grateful.

– [Rafs](https://dba.stackexchange.com/users/93601/rafs)

[May 30, 2022 at 16:34](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment609250_220937)

* @RTD Well, I tried to find that command in the manual. But searching for “TABLE” in Postgres doc results in an endless number of hits.

– [Basil Bourque](https://dba.stackexchange.com/users/19079/basil-bourque)

[May 30, 2022 at 17:15](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment609252_220937)

* @RTD Found it, after I thought to use the Index chapter of the doc. The TABLE command is documented on the page for [SELECT](https://www.postgresql.org/docs/current/sql-select.html). Search that page for “TABLE Command”, about 3/4 down the page.

– [Basil Bourque](https://dba.stackexchange.com/users/19079/basil-bourque)

[May 30, 2022 at 17:25](https://dba.stackexchange.com/questions/220933/query-for-all-the-postgres-configuration-parameters-current-values#comment609254_220937)

<https://materialize.com/docs/connect-sources/cdc-postgres-direct/#:~:text=For%20CDC%2C%20you'll%20need,all%20changes%20can%20take%20effect>.

For CDC, you'll need to set it to logical in the database configuration file ( postgresql. conf ). Keep in mind that changing the wal\_level requires a restart of the Postgres instance and can affect database performance. Restart the database so all changes can take effect.

# **PostgreSQL CDC**

Change Data Capture (CDC) allows you to track and propagate changes in a Postgres database to downstream consumers based on its Write-Ahead Log (WAL).

This guide shows you how to propagate CDC data directly from [Postgres](https://materialize.com/docs/sql/create-source/postgres) to Materialize using Postgres’ native replication protocol.

## Database setup

**Minimum requirements:** PostgreSQL 10+

Before creating a source in Materialize, you need to ensure that the upstream database is configured to support [logical replication](https://www.postgresql.org/docs/10/logical-replication.html).

* [Self-hosted](https://materialize.com/docs/connect-sources/cdc-postgres-direct/#self-hosted-t0)
* [AWS RDS](https://materialize.com/docs/connect-sources/cdc-postgres-direct/#aws-rds-t0)
* [AWS Aurora](https://materialize.com/docs/connect-sources/cdc-postgres-direct/#aws-aurora-t0)
* [Azure DB](https://materialize.com/docs/connect-sources/cdc-postgres-direct/#azure-db-t0)
* [Cloud SQL](https://materialize.com/docs/connect-sources/cdc-postgres-direct/#cloud-sql-t0)

As a superuser:

1. Check the [wal\_level configuration](https://www.postgresql.org/docs/current/wal-configuration.html) setting:
2. SHOW wal\_level;

The default value is replica. For CDC, you’ll need to set it to logical in the database configuration file (postgresql.conf). Keep in mind that changing the wal\_level requires a restart of the Postgres instance and can affect database performance.

1. Restart the database so all changes can take effect.

Once logical replication is enabled:

1. Grant the required privileges to the replication user:
2. ALTER ROLE "dbuser" WITH REPLICATION;

And ensure that this user has the right permissions on the tables you want to replicate:

GRANT CONNECT ON DATABASE dbname TO dbuser;

GRANT USAGE ON SCHEMA schema TO dbuser;

GRANT SELECT ON ALL TABLES IN SCHEMA schema TO dbuser;

**Note:** SELECT privileges on the tables you want to replicate are needed for the initial table sync.

1. Set the replica identity to FULL for the tables you want to replicate:
2. ALTER TABLE repl\_table REPLICA IDENTITY FULL;

This setting determines the amount of information that is written to the WAL in UPDATE and DELETE operations.

As a heads-up, you should expect a performance hit in the database from increased CPU usage. For more information, see the [PostgreSQL documentation](https://www.postgresql.org/docs/current/logical-replication-publication.html).

1. Create a [publication](https://www.postgresql.org/docs/current/logical-replication-publication.html) with the tables you want to replicate:

For specific tables:

CREATE PUBLICATION mz\_source FOR TABLE table1, table2;

For all tables in Postgres:

CREATE PUBLICATION mz\_source FOR ALL TABLES;

The mz\_source publication will contain the set of change events generated from the specified tables, and will later be used to ingest the replication stream. We strongly recommend that you **limit** publications only to the tables you need.

## Create a source

Postgres sources ingest the raw replication stream data for all tables included in a publication to avoid creating multiple [replication slots](https://materialize.com/docs/sql/create-source/postgres/#postgresql-replication-slots) and minimize the required bandwidth.

When you define a Postgres source, Materialize will automatically create a **subsource** for each original table in the publication (so you don’t have to!):

Create subsources for all tables included in the Postgres publication

CREATE SOURCE mz\_source

FROM POSTGRES CONNECTION pg\_connection (PUBLICATION 'mz\_source')

FOR ALL TABLES

WITH (SIZE = '3xsmall');

Create subsources for all tables in specific schemas included in the Postgres publication

CREATE SOURCE mz\_source

FROM POSTGRES CONNECTION pg\_connection (PUBLICATION 'mz\_source')

FOR SCHEMAS (public, project)

WITH (SIZE = '3xsmall');

Create subsources for specific tables included in the Postgres publication

CREATE SOURCE mz\_source

FROM POSTGRES CONNECTION pg\_connection (PUBLICATION 'mz\_source')

FOR TABLES (table\_1, table\_2 AS alias\_table\_2)

WITH (SIZE = '3xsmall');

**NOTE:**Materialize performs an initial sync of all tables in the publication before it starts ingesting change events.

## Create a materialized view

Any materialized view that depends on replication subsources will be incrementally updated as change events stream in, as a result of INSERT, UPDATE and DELETE operations in the original Postgres database.

CREATE MATERIALIZED VIEW cnt\_view1 AS

SELECT field1,

COUNT(\*) AS cnt

FROM view1

GROUP BY field1;