

# Резервное копирование Архив журналов предзаписи



## **Авторские права**

© Postgres Professional, 2018–2022

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

## **Использование материалов курса**

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## **Обратная связь**

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## **Отказ от ответственности**

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

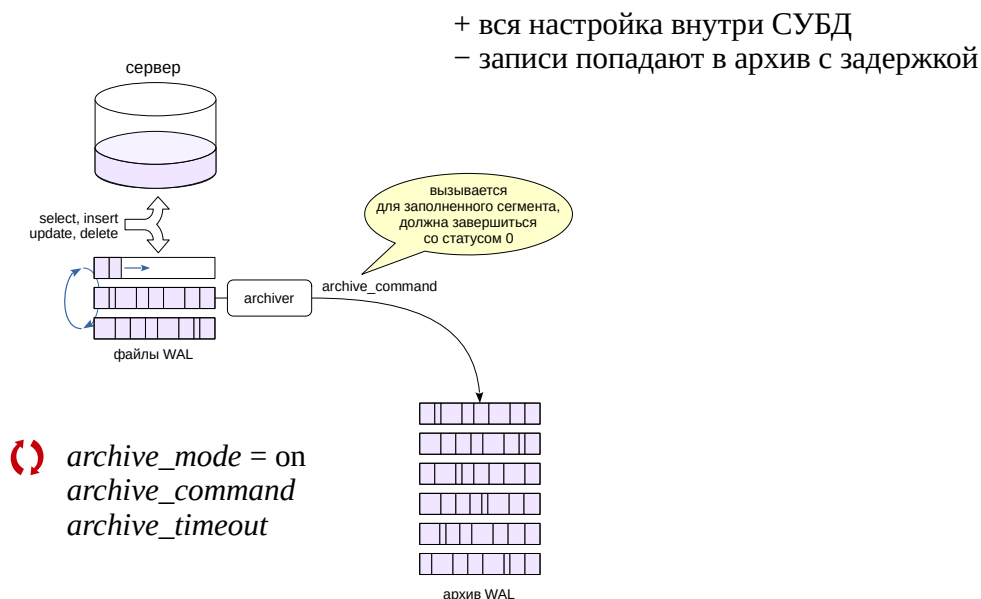
Файловый архив — непрерывная архивация

Потоковый архив — утилита `pg_receivewal`

Восстановление с использованием архива

Очистка архива

# Непрерывная архивация



3

Раз у нас есть базовая резервная копия и журнал предзаписи, то, добавляя каким-то образом к копии все новые журнальные файлы, генерируемые сервером, мы можем восстановить систему не только на момент копирования файловой системы, но и вообще на произвольный момент времени.

И такая возможность есть. Но журнальные записи добавляются не к самой резервной копии, а в отдельный «архив».

Отправка журнальных файлов в архив реализуется фоновым процессом `archiver`, который включается параметром `archive_mode = on`.

Для копирования определяется произвольная команда shell в параметре `archive_command`. Она вызывается при заполнении очередного сегмента WAL. Если команда завершается с нулевым статусом, то считается, что сегмент успешно помещен в архив и может быть удален с сервера. При ненулевом статусе этот сегмент (и следующие за ним) не будут удаляться, а сервер будет периодически повторять команду архивирования, пока не получит 0.

Параметр `archive_timeout` позволяет указать максимальное время переключения на новый сегмент WAL — это позволяет при невысокой активности сервера сохранять тем не менее журналы не реже, чем хотелось бы (иными словами, потерять данные максимум за указанное время). Переключение на новый сегмент можно выполнить и вручную с помощью функции `pg_switch_wal()`.

<https://postgrespro.ru/docs/postgresql/13/continuous-archiving>

## Команда ОС, архивирующая заполненный сегмент WAL

копирует файл %p в архив под именем %f,  
сам архив может быть организован любым образом  
должна завершаться со статусом 0 только при успехе  
(пустая команда не считается успешной и приостанавливает архивацию)  
не должна перезаписывать уже существующие файлы  
должна гарантировать запись в энергонезависимую память  
удобно реализовать нужную логику в собственном скрипте

## Мониторинг

текущий статус — представление `pg_stat_archiver`  
удобно включить коллектор сообщений (`logging_collector = on`)  
и получать диагностику в журнале сообщений сервера

4

Команда архивирования должна скопировать указанный файл в некий архив. Архив может быть организован любым образом. Например, это может быть файловая система на отдельном сервере.

Команда обязана завершаться с нулевым статусом только в случае успеха.

Команда не должна перезаписывать уже существующие файлы, так как это скорее всего означает какую-то ошибку, и перезапись файла может погубить архив.

Для гарантии надежности команда должна обеспечить попадание файла в энергонезависимую память: иначе при сбое архива можно потерять сегмент, если сервер PostgreSQL успеет его удалить.

Текущий статус архивации показывает представление `pg_stat_archiver`. Удобно включить сбор сообщений с помощью процесса `logging_collector`, так как в этом случае сообщения об ошибках при выполнении `archive_command` будут попадать в журнал сервера.

Если подразумевается сложная логика, то ее удобно записать в скрипт и использовать имя скрипта в качестве команды копирования.

<https://postgrespro.ru/docs/postgresql/13/runtime-config-wal.html#GUC-ARCHIVE-COMMAND>

## Настройка непрерывной архивации

Архив будем хранить в каталоге /var/lib/postgresql/archive. Он должен быть доступен пользователю-владельцу PostgreSQL.

```
student$ sudo mkdir /var/lib/postgresql/archive
```

```
student$ sudo chown postgres /var/lib/postgresql/archive
```

В реальной практике архив может размещаться на отдельном сервере или дисковой системе с доступом по сети.

Включим режим архивирования и установим команду копирования заполненных сегментов журнала.

```
α=> \c - postgres
```

You are now connected to database "student" as user "postgres".

```
α=> ALTER SYSTEM SET archive_mode = on;
```

ALTER SYSTEM

В archive\_command мы сначала проверяем наличие файла с указанным именем в архиве, и копируем его только в случае отсутствия.

```
α=> ALTER SYSTEM SET archive_command = 'test ! -f /var/lib/postgresql/archive/%f && cp %p /var/lib/postgresql/archive/%f';
```

ALTER SYSTEM

В archive\_command можно указать произвольную команду, лишь бы она завершалась со статусом 0 только в случае успеха. Например, можно организовать сжатие архивируемых сегментов:

```
α=> -- SET archive_command = 'test ! -f /var/lib/postgresql/archive/%f && gzip <%p >/var/lib/postgresql/archive/%f';
```

В идеале команда архивирования должна выполнять sync, чтобы файл гарантированно попал в энергонезависимую память. Иначе при сбое он может пропасть из архива, а сервер может успеть стереть его из каталога pg\_wal.

В общем случае удобно поместить всю необходимую логику архивирования в отдельный скрипт и вызывать его:

```
α=> -- SET archive_command = 'archive.sh "%f" "%p"';
```

Изменение archive\_mode требует рестарта сервера.

```
student$ sudo pg_ctlcluster 13 alpha restart
```

Проверим работу архивации. Создадим базу и таблицу.

```
student$ psql
```

```
α=> CREATE DATABASE backup_archive;
```

CREATE DATABASE

```
α=> \c backup_archive
```

You are now connected to database "backup\_archive" as user "student".

```
α=> CREATE TABLE t(s text);
```

CREATE TABLE

```
α=> INSERT INTO t VALUES ('Привет, мир!');
```

INSERT 0 1

Вот какой сегмент WAL используется сейчас:

```
α=> SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
      pg_walfile_name
-----
000000010000000000000003
(1 row)
```

Обратите внимание: первые восемь цифр в имени файла — номер текущей линии времени.

Чтобы заполнить файл, пришлось бы выполнить большое количество операций. В тестовых целях проще принудительно переключить сегмент:

```
student$ psql -U postgres -c "SELECT pg_switch_wal();"
```

```
      pg_switch_wal
-----
0/301C460
(1 row)
```

```
α=> INSERT INTO t VALUES ('Доброе утро, страна!');
```

INSERT 0 1

Сегмент сменился:

```

α=> SELECT pg_walfile_name(pg_current_wal_lsn());

      pg_walfile_name
-----
00000001000000000000000004
(1 row)

```

А предыдущий должен был попасть в архив. Проверим:

```

student$ sudo ls -l /var/lib/postgresql/archive

total 16384
-rw----- 1 postgres postgres 16777216 янв 16 12:16 000000010000000000000003

```

Текущий статус архивации показывает представление pg\_stat\_archiver:

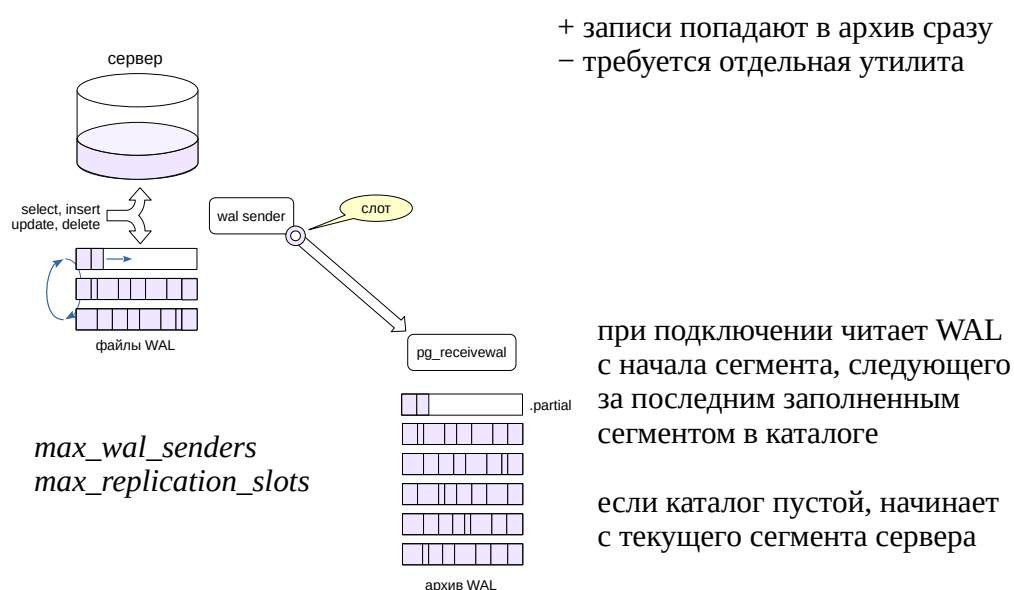
```

α=> SELECT * FROM pg_stat_archiver \gx

-[ RECORD 1 ]-----+-----
archived_count      | 1
last_archived_wal   | 000000010000000000000003
last_archived_time  | 2024-01-16 12:16:14.241057+03
failed_count        | 0
last_failed_wal     |
last_failed_time    |
stats_reset         | 2024-01-16 12:16:06.643908+03

```

Таким образом, файловая архивация настроена и работает.



Можно организовать пополнение архива иным способом, с помощью протокола репликации. Для этого используется утилита `pg_receivewal`.

Обычно утилита запускается на отдельном сервере и подключается к серверу с параметрами, указанными в ключах. Утилита может (и должна) использовать слот репликации, чтобы гарантированно не потерять записи.

Утилита формирует файлы аналогично тому, как это делает сам сервер, и записывает их в указанный каталог. Еще не до конца заполненные сегменты отличаются префиксом `.partial`. Синхронизация с файловой системой по умолчанию происходит только при закрытии файла-сегмента.

Архивирование всегда начинается с начала сегмента, следующего за последним уже полностью заполненным сегментом, который присутствует в архиве. Если архив пуст (первый запуск), архивирование начинается с начала текущего сегмента. Вместе со слотом это гарантирует отсутствие пропусков в архиве, даже если утилита отключалась на некоторое время. Утилита будет получать записи бесконечно, но, если указан ключ `--endpos=lsn`, она завершит работу по получении записи с заданным LSN.

Требуется учесть, что сама по себе утилита не запускается автоматически (как сервис) и не демонизируется. Она потребует дополнительного мониторинга и (в случае репликации) действий по переключению на другой сервер.

<https://postgrespro.ru/docs/postgresql/13/app-pgreceivewal>

## Потоковый архив

Чтобы настроить пополнение архива по протоколу потоковой репликации, остановим сначала файловую архивацию.

```
α=> \c - postgres
```

You are now connected to database "backup\_archive" as user "postgres".

```
α=> ALTER SYSTEM RESET archive_mode;
```

ALTER SYSTEM

```
α=> ALTER SYSTEM RESET archive_command;
```

ALTER SYSTEM

```
student$ sudo pg_ctlcluster 13 alpha restart
```

Текущее состояние архива.

```
student$ sudo ls -l /var/lib/postgresql/archive
```

```
total 32768
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000003
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000004
```

Сначала попросим утилиту pg\_receivewal создать слот, чтобы гарантировать получение всех записей журнала.

```
postgres$ pg_receivewal --create-slot --slot=archive
```

Затем запустим ее фоном в режиме архивации. Увидев, что в архиве уже есть файлы, утилита запросит у сервера следующий сегмент, чтобы в архиве не было пропусков.

```
postgres$ pg_receivewal -D /var/lib/postgresql/archive --slot=archive
```

Добавим в таблицу много строк и удалим их.

```
student$ psql -d backup_archive
```

```
α=> INSERT INTO t SELECT 'И снова здравствуйте.' FROM generate_series(1,200000);
```

INSERT 0 200000

```
α=> DELETE FROM t WHERE s = 'И снова здравствуйте.';
```

DELETE 200000

```
α=> VACUUM t;
```

VACUUM

В архиве появились новые файлы.

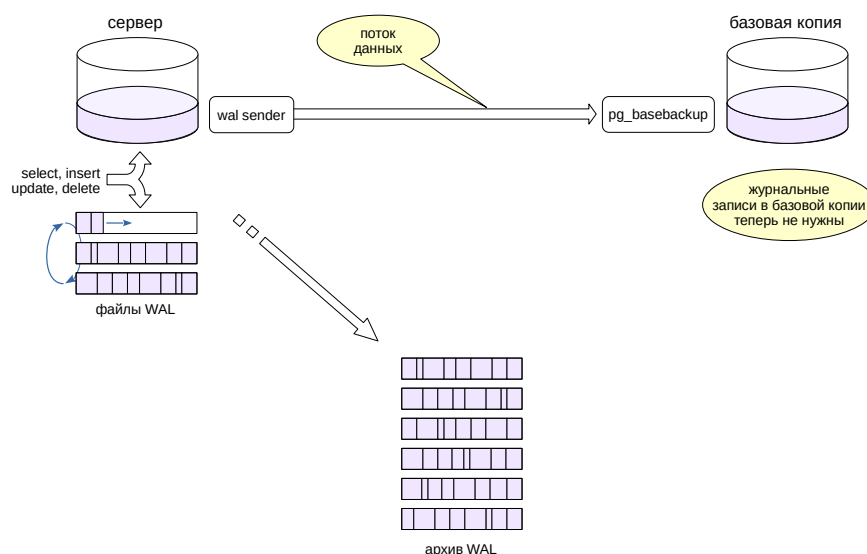
```
student$ sudo ls -l /var/lib/postgresql/archive
```

```
total 65536
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000003
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000004
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000005
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000006.partial
```

Последний файл, скорее всего, имеет суффикс .partial — в него идет запись.

Теперь мы настроили потоковую архивацию.





Если мы располагаем настроенным архивом журнала предзаписи, то в базовой резервной копии файлы журнала становятся не обязательны — ведь их при восстановлении можно получить из архива. Поэтому `pg_basebackup` можно запускать с ключом `--wal-method=none`.

(Но и никакого вреда от журнальных файлов, кроме занимаемого объема, внутри базовой копии тоже нет. Более того, они позволяют восстановить систему в ситуации, если архив окажется недоступен.)

## Базовая резервная копия

Поскольку архив пополняется автоматически, попросим pg\_basebackup не добавлять файлы журнала к резервной копии:

```
student$ pg_basebackup --wal-method=none --pgdata=/home/student/backup
```

NOTICE: WAL archiving is not enabled; you must ensure that all required WAL segments are copied through other means to complete the backup

Каталог pg\_wal резервной копии пуст:

```
student$ ls -l /home/student/backup/pg_wal/
```

```
total 4
drwx----- 2 student student 4096 янв 16 12:16 archive_status
```

А в архиве прибавилось файлов:

```
student$ sudo ls -l /var/lib/postgresql/archive
```

```
total 81920
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000003
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000004
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000005
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000006
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000007
```

Заглянем в сгенерированный файл метки:

```
student$ cat /home/student/backup/backup_label
```

```
START WAL LOCATION: 0/7000028 (file 00000001000000000000000007)
CHECKPOINT LOCATION: 0/7000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2024-01-16 12:16:20 MSK
LABEL: pg_basebackup base backup
START TIMELINE: 1
```

Главная информация в этом файле — номер линии времени (строка START TIMELINE) и указание начальной точки для восстановления (START WAL LOCATION). Теоретически восстановление можно начать и с более ранней (но не более поздней) позиции, но это потребует больше времени.

`tablespace_map`

пути для табличных пространств

`backup_label`

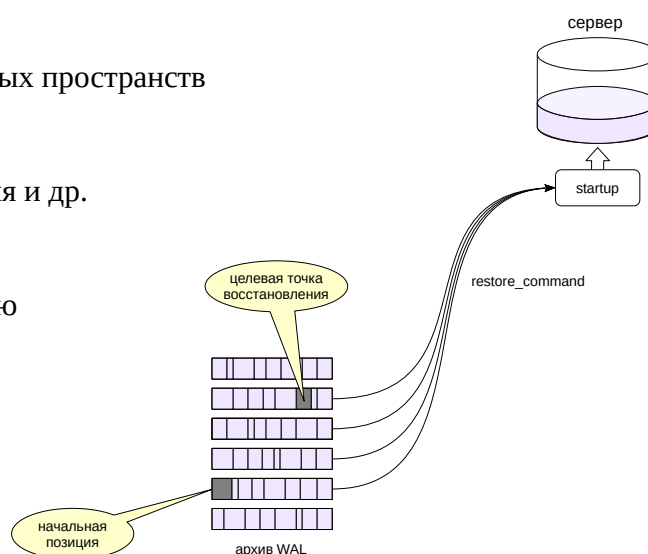
начальная позиция и др.

`recovery.signal`

создается вручную

`postgresql.conf`

`restore_command`  
целевая точка  
и др.



Процессом восстановления управляют три файла и параметры.

Файл **tablespace\_map** генерируется при создании базовой резервной копии и содержит пути для табличных пространств. Если символические ссылки в `pg_tblspc` отсутствуют (это верно для формата `tar`), то они будут созданы при старте сервера на основании информации в `tablespace_map`.

С этим файлом мы уже встречались в теме «Базовая резервная копия».

Файл метки **backup\_label** также генерируется автоматически при создании базовой резервной копии и содержит название, время создания копии и — самое важное — номер линии времени, название сегмента WAL и позицию в нем, с которой надо начинать восстановление.

Третий файл, **recovery.signal**, создается вручную. Его наличие дает указание серверу перейти в режим восстановления. Содержимое файла игнорируется. Если `recovery.signal` отсутствует, то PostgreSQL считает, что он выполняет автоматическое восстановление после сбоя. Если же файл есть, сервер понимает, что происходит управляемое пользователем восстановление из резервной копии.

В версиях PostgreSQL до 11 параметры восстановления задавались в отдельном конфигурационном файле `recovery.conf`. Начиная с версии 12, параметры восстановления задаются так же, как и все остальные.

По умолчанию проигрываются все доступные журнальные записи

## Восстановление до определенной точки (PITR)

<code>recovery_target = 'immediate'</code>	только восстановить согласованность
<code>recovery_target_name = 'имя'</code>	до именованной точки, созданной <code>pg_create_restore_point('имя')</code>
<code>recovery_target_time = 'время'</code>	до указанного времени
<code>recovery_target_xid = 'xid'</code>	до указанной транзакции
<code>recovery_target_lsn = 'lsn'</code>	до указанного LSN
<code>recovery_target_inclusive = on off</code>	включать ли указанную точку

Если не задана цель восстановления (один из параметров `recovery_target_*`), базы данных будут восстановлены максимально близко к моменту сбоя. Однако процесс восстановления можно остановить и в любой другой момент, указав **целевую точку**.

Параметр `recovery_target = 'immediate'` остановит восстановление, как только будет достигнута согласованность. Фактически, это эквивалентно восстановлению из базовой копии без архива.

Параметр `recovery_target_name` позволяет указать именованную точку восстановления, созданную ранее с помощью функции `pg_create_restore_point()`. Это полезно, если заранее известно, что может потребоваться восстановление.

Параметр `recovery_target_time` позволяет указать произвольное время (timestamp), `recovery_target_xid` — произвольную транзакцию, `recovery_target_lsn` — номер LSN. С помощью параметра `recovery_target_inclusive` можно включить или исключить саму точку.




<https://postgrespro.ru/docs/postgresql/13/runtime-config-wal#RUNTIME-CO-NFIG-WAL-ARCHIVE-RECOVERY>

## Команда восстановления

*restore\_command* — команда, «обратная» *archive\_command*: копирует файл %f из архива в %p  
должна завершаться со статусом 0 только при успехе

## По окончании восстановления (если задана цель)

*recovery\_target\_action* =

<b>pause</b>		завершить — <code>pg_wal_replay_resume()</code> задать новую цель + рестарт
promote		принимает подключения
shutdown		останавливается

Для восстановления необходимо задать параметр *restore\_command*, который определяет команду, «обратную» команде архивирования *archive\_command*. Она должна скопировать файл из архива в каталог `pg_wal` и завершаться со статусом 0 только при успехе. Это очень важно, так как в процессе восстановления сервер может выполнять команду для файлов, которых не окажется в архиве — это не ошибка, но команда должна завершиться с ненулевым статусом.

<https://postgrespro.ru/docs/postgresql/13/runtime-config-wal#RUNTIME-CONFIG-WAL-ARCHIVE-RECOVERY>

Если задана цель, параметр *recovery\_target\_action* позволяет выбрать состояние, в котором окажется сервер по окончании восстановления.

По умолчанию (значение `pause`) процедура восстановления приостанавливается. Администратор может выполнить запросы к данным и решить, корректно ли была выбрана цель, после чего либо завершить восстановление, вызвав функцию `pg_wal_replay_resume()`, либо задать новую цель, перезапустить сервер и продолжить применение журнальных записей.

Если задано значение `promote`, сервер сразу сможет принимать запросы на подключения.

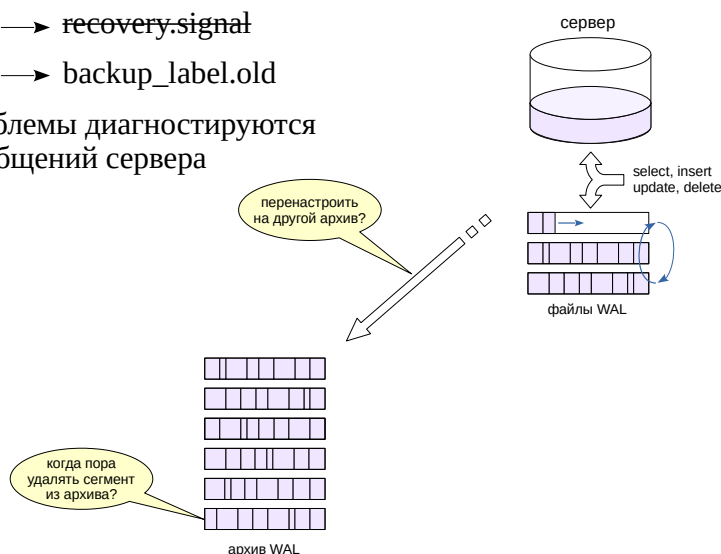
Значение `shutdown` приведет к остановке сервера. При этом файл `recovery.signal` не удаляется.

# Окончание восстановления

recovery.signal → ~~recovery.signal~~

backup\_label → backup\_label.old

возможные проблемы диагностируются  
по журналу сообщений сервера



13

После того, как восстановление закончено, процесс startup завершается, postmaster запускает остальные служебные процессы, необходимые для работы экземпляра, и сервер начинает работать в обычном режиме. Файл `recovery.signal` удаляется, `backup_label` переименовывается в `backup_label.old`.

Возможна ситуация, когда запущенный сервер не стартует. В этом случае в операционной системе будут отсутствовать процессы (и не будет файла `postmaster.pid`), а `recovery.signal` не будет удален. Причину ошибки можно узнать из журнала сообщений сервера (например, в параметрах была допущена ошибка). После исправления причины ошибки сервер следует запустить еще раз.

Важный момент: если на сервере-источнике было настроено непрерывное архивирование, то на резервном сервере его надо либо отключить, либо перенаправить в другой архив.

## Восстановление из базовой резервной копии

Зададим настройки восстановления на сервере beta (до версии 12 их нужно помещать в отдельный файл `recovery.conf`). В простейшем случае достаточно указать команду восстановления, которая будет копировать указанный сегмент WAL обратно из архива по указанному пути:

```
student$ echo "restore_command = 'cp /var/lib/postgresql/archive/%f %p'" >>/home/student/backup/postgresql.auto.conf
```

Если не указана целевая точка восстановления (один из параметров `recovery_target*`), то к базовой резервной копии будут применены записи WAL из всех файлов в архиве.

Наличие файла `recovery.signal` — указание серверу при старте войти в режим управляемого восстановления:

```
student$ touch /home/student/backup/recovery.signal
```

Выкладываем резервную копию в каталог данных сервера beta и запускаем его.

```
student$ sudo pg_ctlcluster 13 beta status
```

Error: /var/lib/postgresql/13/beta is not accessible or does not exist

```
student$ sudo rm -rf /var/lib/postgresql/13/beta
```

```
student$ sudo mv /home/student/backup /var/lib/postgresql/13/beta
```

```
student$ sudo chown -R postgres /var/lib/postgresql/13/beta
```

```
student$ sudo pg_ctlcluster 13 beta start
```

После успешного восстановления файл `recovery.signal` удаляется, а файл метки переименовывается в `backup_label.old`:

```
student$ sudo ls -l /var/lib/postgresql/13/beta | egrep 'recovery.*|backup_label.*'
```

```
-rw----- 1 postgres student    224 янв 16 12:16 backup_label.old
```

Проверим, что восстановлено в таблице:

```
student$ psql -p 5433 -d backup_archive
```

```
β=> SELECT * FROM t;
```

```
      s
-----
Привет, мир!
Доброе утро, страна!
(2 rows)
```

Порядковый номер, +1 при каждом восстановлении

номер  $N$  линии времени входит в имя сегмента WAL

история сохраняется в файле `pg_wal/N.history` и архивируется

`recovery_target_timeline = 'latest'`

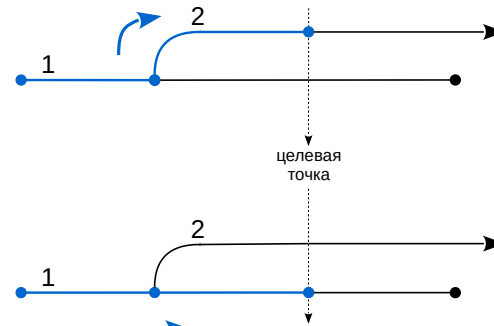
восстановление по последней линии  
(по умолчанию)

`recovery_target_timeline = 'current'`

восстановление по текущей линии

`recovery_target_timeline = '1'`

восстановление по указанной линии



15

После восстановления на момент в прошлом сервер начинает генерировать новые сегменты WAL, которые будут пересекаться с сегментами «из прошлой жизни». Чтобы не потерять сегменты и, вместе с ними, возможность восстановления на другой момент, PostgreSQL вводит понятие линии времени. После каждого восстановления с использованием `recovery.signal` номер линии времени увеличивается, и этот номер является частью номера сегментов WAL.

Линии времени образуют древовидную структуру, на рисунках приведен пример. Первая линия имеет номер 1. Произошло восстановление на момент времени в прошлом, и началась линия 2.

Если выполнить восстановление с указанием параметра `recovery_target_timeline = 'latest'` (по умолчанию с версии 12), то, дойдя до точки ветвления, применение записей WAL продолжится по второй (более поздней) линии.

Если же указать параметр `recovery_target_timeline = '1'`, то восстановление продолжится по первой (указанной) линии, а если указать `recovery_target_timeline = 'current'`, то по текущей (в нашем случае тоже по линии 1, это поведение по умолчанию до версии 11).

Информацию о точках ветвления PostgreSQL берет из файлов истории `pg_wal/N.history`. Поэтому их никогда не следует удалять из архива.



## Линии времени

Что стало с линией времени после восстановления?

```
β=> SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
pg_walfile_name
-----
000000020000000000000008
(1 row)
```

Номер увеличился на единицу.

В каталоге pg\_wal появился файл истории, соответствующий этой ветви:

```
postgres$ ls -l /var/lib/postgresql/13/beta/pg_wal/00000002.history
```

```
-rw----- 1 postgres student 41 янв 16 12:16 /var/lib/postgresql/13/beta/pg_wal/00000002.history
```

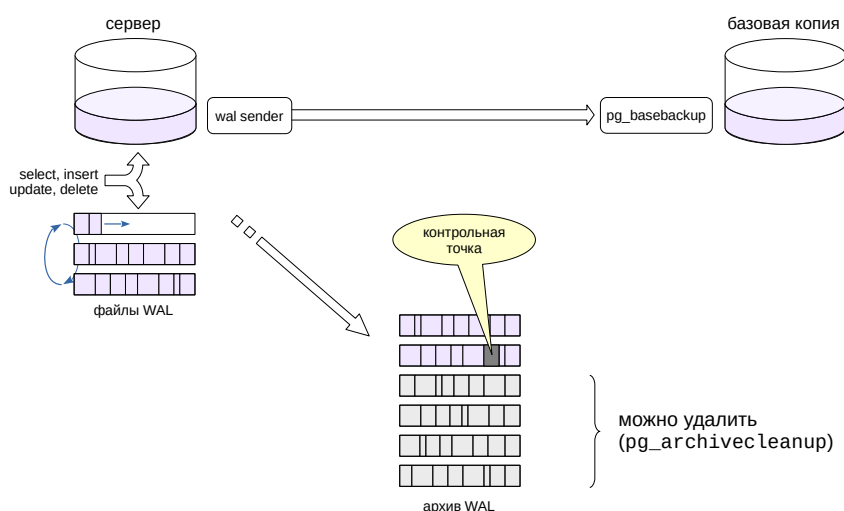
В нем есть информация о «точках ветвления», через которые мы пришли в данную линию времени:

```
student$ sudo cat /var/lib/postgresql/13/beta/pg_wal/00000002.history
```

```
1          0/8000000          no recovery target specified
```

Эти файлы PostgreSQL использует, когда мы указываем линию времени в параметре `recovery_target_timeline`. Поэтому файлы истории подлежат архивации вместе с сегментами WAL, и удалять из архива их не надо.

# Очистка архива



17

Архив со временем разрастается, поэтому требуется периодическая очистка от ненужных файлов.

Для очистки можно воспользоваться утилитой **pg\_archivecleanup**. Утилите нужно указать имя сегмента WAL, и она удалит все ему предшествующие.

<https://postgrespro.ru/docs/postgresql/13/pgarchivecleanup>

Вопрос в том, как определить позицию в архиве, начиная с которой сегменты уже не нужны.

Если архив используется только для восстановления одного резервного экземпляра, можно выполнять очистку сразу после восстановления. Для этого служит параметр *recovery\_end\_command*, в котором задается команда (например, `pg_archivecleanup`), которая выполнится по окончании восстановления. В команде можно использовать комбинацию `%r`, которая заменится на имя файла, содержащего запись о последней выполненной контрольной точке.

Но если архив должен поддерживать восстановление из нескольких базовых резервных копий, то для автоматизации очистки либо придется написать собственный скрипт, либо — что лучше — воспользоваться одной из сторонних программ резервного копирования, таких как `pg_rbackuper`, которые позволяют гибко настроить политику хранения резервных копий.

## Очистка архива

Сейчас архив первого сервера содержит ненужные файлы, они попали туда до формирования базовой копии:

```
student$ sudo ls -l /var/lib/postgresql/archive
```

```
total 98304
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000003
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000004
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000005
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000006
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000007
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000008.partial
```

Утилите `pg_archivecleanup` нужно передать путь к архиву и имя последнего сохраняемого сегмента WAL (его можно найти в файле `backup_label`).

```
student$ sudo head -n 1 /var/lib/postgresql/13/beta/backup_label.old
```

```
START WAL LOCATION: 0/7000028 (file 00000001000000000000000007)
```

```
student$ sudo pg_archivecleanup /var/lib/postgresql/archive 00000001000000000000000007
```

Посмотрим, какие файлы остались в архиве:

```
student$ sudo ls -l /var/lib/postgresql/archive
```

```
total 32768
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000007
-rw----- 1 postgres postgres 16777216 янв 16 12:16 00000001000000000000000008.partial
```

Архив очищен.

Чтобы архив очищался по окончании восстановления, можно задать параметр

```
α=> -- SET recovery_end_command = 'pg_archivecleanup /var/lib/postgresql/archive %r'
```

## Физическое резервирование — базовые резервные копии и архив журнала предзаписи

### Хорошо подходит

- для текущего периодического резервирования
- для восстановления после сбоя с минимальной потерей данных
- для восстановления на произвольный момент времени
- для резервирования данных большого объема

### Плохо подходит

- для длительного хранения

### Не подходит

- для миграции на другую платформу

1. На первом сервере создайте базу данных и в ней таблицу с какими-нибудь данными.
2. Настройте непрерывное архивирование.
3. Сделайте базовую резервную копию кластера с помощью `pg_basebackup`, без файлов журнала.
4. Вставьте еще несколько строк в таблицу и убедитесь, что текущий сегмент WAL попал в архив.
5. Восстановите второй сервер из резервной копии, указав в параметрах одну только команду восстановления. Проверьте, что в таблице восстановились все строки.
6. Остановите второй сервер и восстановите его повторно из той же резервной копии, на этот раз указав целевую точку восстановления `immediate`. Проверьте таблицу.

## 1. База данных и таблица

```
α=> CREATE DATABASE backup_archive;
CREATE DATABASE
α=> \c backup_archive
You are now connected to database "backup_archive" as user "student".
α=> CREATE TABLE t(s text);
CREATE TABLE
α=> INSERT INTO t VALUES ('Привет, мир!');
INSERT 0 1
```

## 2. Настройка непрерывной архивации

```
student$ sudo mkdir /var/lib/postgresql/archive
student$ sudo chown postgres /var/lib/postgresql/archive
α=> \c - postgres
You are now connected to database "backup_archive" as user "postgres".
α=> ALTER SYSTEM SET archive_mode = on;
ALTER SYSTEM
α=> ALTER SYSTEM SET archive_command = 'test ! -f /var/lib/postgresql/archive/%f && cp %p /var/lib/postgresql/archive/%f';
ALTER SYSTEM
α=> \q
student$ sudo pg_ctlcluster 13 alpha restart
```

## 3. Базовая резервная копия

Поскольку нам потребуется восстанавливаться из одной копии два раза, сделаем ее в формате tar.

```
student$ mkdir /home/student/backup
student$ pg_basebackup --wal-method=none --format=tar --pgdata=/home/student/backup
NOTICE: all required WAL segments have been archived
student$ ls -l /home/student/backup
total 40416
-rw----- 1 student student 219741 янв 16 12:22 backup_manifest
-rw----- 1 student student 41160192 янв 16 12:22 base.tar
```

## 4. Добавление строк в таблицу

```
student$ psql -d backup_archive
α=> INSERT INTO t VALUES ('Еще одна строка');
INSERT 0 1
α=> \c - postgres
```

You are now connected to database "backup\_archive" as user "postgres".

Переключаем сегмент:

```
α=> SELECT pg_walfile_name(pg_current_wal_lsn()), pg_switch_wal();

 pg_walfile_name | pg_switch_wal
-----+-----
 000000010000000000000005 | 0/50026D0
(1 row)
```

Проверяем статус архивации:

```
α=> SELECT last_archived_wal FROM pg_stat_archiver;

 last_archived_wal
-----
 000000010000000000000005
(1 row)
```

## 5. Восстановление из базовой резервной копии

```
student$ sudo pg_ctlcluster 13 beta stop
```

Error: /var/lib/postgresql/13/beta is not accessible or does not exist

```
student$ sudo rm -rf /var/lib/postgresql/13/beta
```

```
student$ sudo mkdir /var/lib/postgresql/13/beta
```

```
student$ sudo tar -xf /home/student/backup/base.tar -C /var/lib/postgresql/13/beta
```

Для простоты отключим непрерывное архивирование на резервном сервере и укажем только команду восстановления:

```
student$ cat << EOF | sudo tee /var/lib/postgresql/13/beta/postgresql.auto.conf
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
EOF
```

```
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
```

Создаем recovery.signal.

```
student$ sudo touch /var/lib/postgresql/13/beta/recovery.signal
```

Запускаем сервер.

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/13/beta
```

```
student$ sudo chmod -R 700 /var/lib/postgresql/13/beta
```

```
student$ sudo pg_ctlcluster 13 beta start
```

```
student$ psql -p 5433 -d backup_archive
```

```
β=> SELECT * FROM t;
```

```

      s
-----
Привет, мир!
Еще одна строка
(2 rows)
```

Восстановились все строки. Поскольку по умолчанию применяются все имеющиеся журнальные записи, сервер сразу перешел в обычный режим и готов принимать запросы на изменение данных:

```
β=> SELECT pg_is_in_recovery();
```

```

pg_is_in_recovery
-----
f
(1 row)
```

## 6. Восстановление из базовой резервной копии — immediate

```
β=> \q
```

```
student$ sudo pg_ctlcluster 13 beta stop
```

```
student$ sudo rm -rf /var/lib/postgresql/13/beta
```

```
student$ sudo mkdir /var/lib/postgresql/13/beta
```

```
student$ sudo tar -xf /home/student/backup/base.tar -C /var/lib/postgresql/13/beta
```

Создаем recovery.signal, в конфигурационный файл записываем команду восстановления и целевую точку:

```
student$ sudo touch /var/lib/postgresql/13/beta/recovery.signal
```

```
student$ cat << EOF | sudo tee /var/lib/postgresql/13/beta/postgresql.auto.conf
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
recovery_target = 'immediate'
EOF
```

```
restore_command = 'cp /var/lib/postgresql/archive/%f %p'
recovery_target = 'immediate'
```

Запускаем сервер.

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/13/beta
```

```
student$ sudo chmod -R 700 /var/lib/postgresql/13/beta
```

```
student$ sudo pg_ctlcluster 13 beta start
```

```
student$ psql -p 5433 -d backup_archive
```

```
β=> SELECT * FROM t;
```

```
      s
-----
Привет, мир!
(1 row)
```

Восстановилась только первая строка.

На этот раз были применены только журнальные записи, необходимые для согласования данных. Сервер уже принимает запросы на чтение, при необходимости восстановление можно продолжить:

```
β=> SELECT pg_is_in_recovery(), pg_is_wal_replay_paused();
```

```
 pg_is_in_recovery | pg_is_wal_replay_paused
-----+-----
t                  | t
(1 row)
```

Поскольку нам не нужно применять последующие записи, завершаем восстановление, переводя сервер в обычный режим (нужны права суперпользователя):

```
β=> \c - postgres
```

```
You are now connected to database "backup_archive" as user "postgres".
```

```
β=> SELECT pg_wal_replay_resume();
```

```
 pg_wal_replay_resume
-----
(1 row)
```

```
β=> SELECT pg_is_in_recovery();
```

```
 pg_is_in_recovery
-----
f
(1 row)
```

Чтобы не выходить из режима восстановления вручную, можно было заранее задать значение `recovery_target_action = 'promote'`.