

# Журналирование Контрольная точка



## Авторские права

© Postgres Professional, 2016–2025

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## 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.

Процесс контрольной точки

Процесс фоновой записи

Мониторинг

Необходимость контрольной точки

Процесс выполнения контрольной точки

Алгоритм восстановления после сбоя

Настройка

## Размер хранимых журнальных записей

с какого сегмента начинать применять журнальные записи при восстановлении?

активно используемая страница может не вытесняться из кеша

## Время восстановления

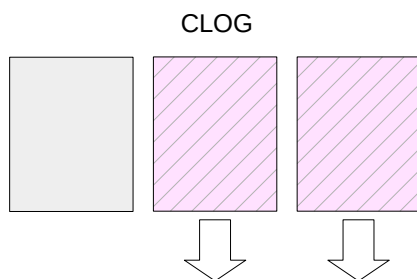
сколько времени займет восстановление после сбоя?

Если не предпринять специальных мер, то активно используемая страница, попав в буферный кеш, может никогда не вытесняться. Это означает, что при восстановлении после сбоя нам придется просматривать *все* журнальные записи, созданные с момента запуска сервера.

На практике это, конечно, недопустимо. Во-первых, файлы занимают много места — их все придется хранить на сервере. Во-вторых, время восстановления будет запредельно большим — придется просмотреть множество журнальных записей.

Поэтому существует процедура **контрольной точки**, которая выполняет сброс всех грязных страниц на диск, но не вытесняет их из кеша. Специальный фоновый процесс checkpointер периодически выполняет контрольную точку. После того как контрольная точка завершена, журнальные записи, предшествующие ее началу, больше не нужны для восстановления.

записываются грязные буферы CLOG

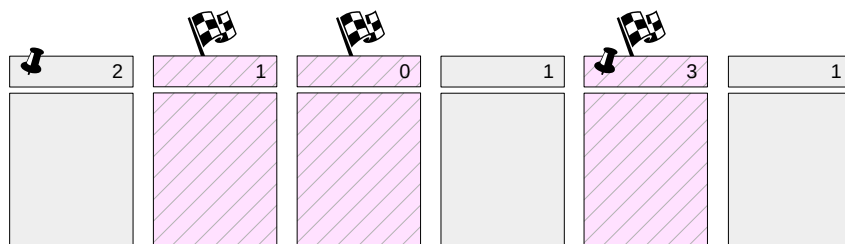


Рассмотрим подробнее, что происходит при выполнении контрольной точки.

Во-первых, помимо буферного кеша, в оперативной памяти располагаются и другие структуры, содержимое которых нужно сохранять на диске.

В частности, контрольная точка сбрасывает на диск буферы статуса транзакций (CLOG). Поскольку количество таких буферов невелико (их 128 штук), они записываются сразу же.

помечаются измененные страницы в буферном кеше



Во-вторых, основная работа контрольной точки — сбросить на диск все страницы из буферного кеша, которые были грязными на момент начала контрольной точки.

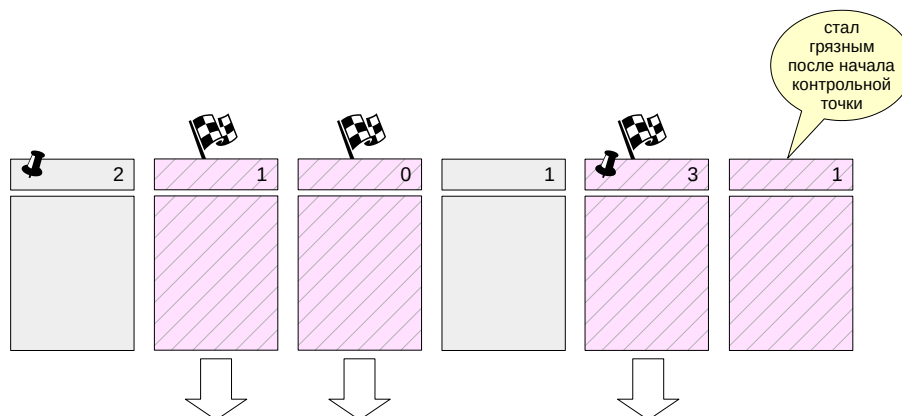
Поскольку размер буферного кеша может быть очень велик, сбрасывать сразу все страницы плохо — это приостановит нормальную работу сервера и создаст большую нагрузку на дисковую подсистему. Поэтому контрольная точка растягивается во времени и фактически превращается в отрезок.

Сначала все измененные на текущий момент страницы помечаются в заголовке буфера специальным флагом...

# Процесс контрольной точки

помеченные страницы постепенно записываются,  
пометка убирается из заголовка буфера

`checkpoint_completion_target = 0.9`



7

...а затем процесс контрольной точки *постепенно* проходит по всем буферам и сбрасывает помеченные страницы на диск. Еще раз отметим, что страницы не вытесняются из кеша, а только записываются. Поэтому контрольная точка не обращает внимания на число обращений к буферу и признак закрепленности.

Помеченные буферы могут также быть записаны и обслуживающими процессами — смотря кто доберется до буфера первым. В любом случае при записи снимается установленный ранее флаг, так что буфер будет записан только один раз.

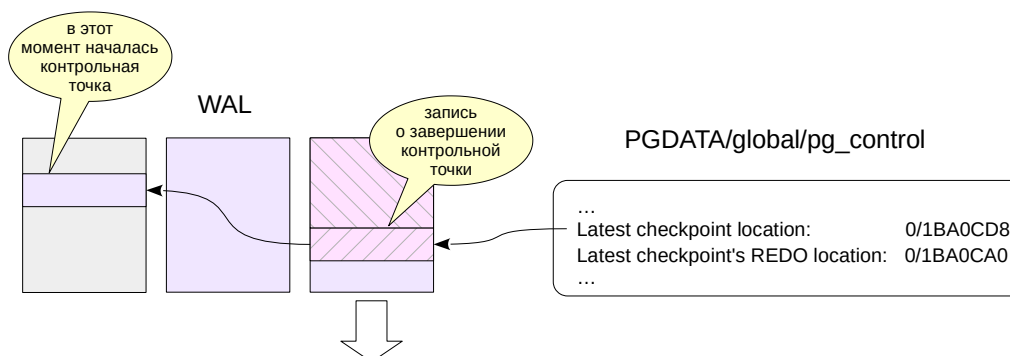
В процессе работы контрольной точки страницы продолжают изменяться в буферном кеше. Но новые грязные буферы уже не рассматриваются процессом контрольной точки, так как на момент начала работы они не были грязными.

Активность записи грязных буферов определяется значением параметра `checkpoint_completion_target`. Он показывает, какую часть времени между двумя соседними контрольными точками следует использовать для записи страниц. Со значением 0.9, заданным по умолчанию, можно ожидать, что PostgreSQL завершит процедуру контрольной точки незадолго до следующей запланированной. Значения выше умолчательного 0.9 использовать не рекомендуется, поскольку фактически процесс может занять несколько больше времени, чем указано в параметре.

# Процесс контрольной точки

в журнале создается запись о завершении контрольной точки с указанием момента ее начала

в файл pg\_control записывается LSN контрольной точки



8

В конце работы процесс создает журнальную запись об окончании контрольной точки. В этой записи содержится LSN момента начала работы контрольной точки. Поскольку контрольная точка ничего не записывает в журнал в начале своей работы, по этому LSN может находиться любая журнальная запись.

Кроме того, в специальный файл PGDATA/global/pg\_control записывается указание на созданную журнальную запись. Таким образом можно быстро выяснить последнюю пройденную контрольную точку.



## Контрольная точка

Заглянем в управляющий файл /var/lib/postgresql/16/main/global/pg\_control. Это можно сделать с помощью утилиты pg\_controldata.

```
student$ sudo /usr/lib/postgresql/16/bin/pg_controldata -D /var/lib/postgresql/16/main
```

```
pg_control version number:      1300
Catalog version number:       202307071
Database system identifier:    7475805087908439375
Database cluster state:       in production
pg_control last modified:     Чт 06 мар 2025 18:04:36
Latest checkpoint location:   0/1BA6478
Latest checkpoint's REDO location: 0/1BA6478
Latest checkpoint's REDO WAL file: 000000010000000000000001
Latest checkpoint's TimeLineID: 1
Latest checkpoint's PrevTimeLineID: 1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:  0:743
Latest checkpoint's NextOID:  16390
Latest checkpoint's NextMultiXactId: 1
Latest checkpoint's NextMultiOffset: 0
Latest checkpoint's oldestXID: 722
Latest checkpoint's oldestXID's DB: 1
Latest checkpoint's oldestActiveXID: 0
Latest checkpoint's oldestMultiXid: 1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid: 0
Latest checkpoint's newestCommitTsXid: 0
Time of latest checkpoint:    Ср 26 фев 2025 22:04:09
Fake LSN counter for unlogged rels: 0/3E8
Minimum recovery ending location: 0/0
Min recovery ending loc's timeline: 0
Backup start location:        0/0
Backup end location:          0/0
End-of-backup record required: no
wal_level setting:            replica
wal_log_hints setting:        off
max_connections setting:      100
max_worker_processes setting: 8
max_wal_senders setting:      10
max_prepared_xacts setting:   0
max_locks_per_xact setting:    64
track_commit_timestamp setting: off
Maximum data alignment:       8
Database block size:          8192
Blocks per segment of large relation: 131072
WAL block size:               8192
Bytes per WAL segment:        16777216
Maximum length of identifiers: 64
Maximum columns in an index:  32
Maximum size of a TOAST chunk: 1996
Size of a large-object chunk:  2048
Date/time type storage:        64-bit integers
Float8 argument passing:       by value
Data page checksum version:    1
Mock authentication nonce:     ad4ea6c69dda94b240abb6ccc86b2382c1d867c9f0ecfb8cd4c807e2aad66201
```

Видим много справочной информации, из которой особый интерес представляют данные о последней контрольной точке и статус кластера: «in production».

Выполним ручную контрольную точку и посмотрим, как это отражается в журнале и в управляющем файле.

```
=> SELECT pg_current_wal_insert_lsn();
```

```
pg_current_wal_insert_lsn
-----
0/1BA64F0
(1 row)
```

```
=> SELECT pg_walfile_name('0/1BA64F0');
```

```

pg_walfile_name
-----
000000010000000000000001
(1 row)

```

=> **CHECKPOINT;**

CHECKPOINT

=> **SELECT pg\_current\_wal\_insert\_lsn();**

```

pg_current_wal_insert_lsn
-----
0/1BA65A0
(1 row)

```

В журнал попадает запись о том, что контрольная точка пройдена (CHECKPOINT\_ONLINE):

=> **CREATE EXTENSION pg\_walinspect;**

CREATE EXTENSION

=> **SELECT start\_lsn, record\_type,**  
**replace(description, ';', E'\n') AS description**  
**FROM pg\_get\_wal\_records\_info('0/1BA64F0', '0/1BA65A0')**  
**WHERE record\_type = 'CHECKPOINT\_ONLINE' \gx**

```

-[ RECORD 1 ]-----
start_lsn   | 0/1BA6528
record_type | CHECKPOINT_ONLINE
description | redo 0/1BA64F0
           | tli 1
           | prev tli 1
           | fpw true
           | xid 0:743
           | oid 16390
           | multi 1
           | offset 0
           | oldest xid 722 in DB 1
           | oldest multi 1 in DB 1
           | oldest/newest commit timestamp xid: 0/0+
           | oldest running xid 743
           | online

```

В описании записи указан LSN начала контрольной точки (redo).

Сравним с данными управляющего файла:

**student\$ sudo /usr/lib/postgresql/16/bin/pg\_controldata -D /var/lib/postgresql/16/main | egrep 'Latest.\*location'**

```

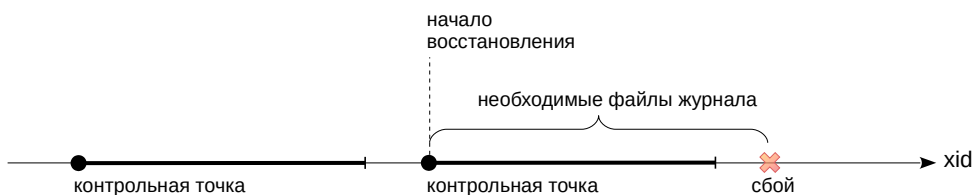
Latest checkpoint location:      0/1BA6528
Latest checkpoint's REDO location: 0/1BA64F0

```

Информация об LSN, очевидно, совпадает.

## При старте сервера после сбоя

1. найти  $LSN_0$  начала последней завершенной контрольной точки
2. применить каждую запись журнала, начиная с  $LSN_0$ , если LSN записи больше, чем LSN страницы
3. оборвать начатые, но незафиксированные транзакции
4. перезаписать нежурналируемые таблицы init-файлами
5. выполнить контрольную точку



10

Если в работе сервера произошел сбой, то при последующем запуске процесс startup обнаруживает это (в файле `pg_control` статус отличен от «shut down») и выполняет автоматическое восстановление.

Сначала процесс читает из того же файла LSN записи о последней завершенной контрольной точке. Из этой записи процесс узнает позицию  $LSN_0$  начала этой контрольной точки.

(Для полноты картины заметим: если присутствует файл `backup_label`, запись о контрольной точке читается из него — это нужно для восстановления из резервных копий. Подробнее см. курс DBA3.)

Далее процесс startup читает журнал вперед от найденной позиции, последовательно применяя записи к страницам, если в этом есть необходимость (что можно проверить, сравнив LSN страницы на диске с LSN журнальной записи). Изменение страниц происходит в буферном кеше, как при обычной работе.

Записи, относящиеся к страницам CLOG, восстанавливают статус транзакций. Транзакции, не зафиксированные к концу восстановления, считаются оборванными; их изменения не видны в снимках данных.

Аналогично записи применяются и к файлам: например, если запись говорит, что файл должен быть создан, а его нет — файл создается.

В конце процесса все нежурналируемые таблицы перезаписываются с помощью образов в init-файлах. На этом процесс startup завершает работу, после чего процесс checkpointer выполняет контрольную точку, чтобы зафиксировать восстановленное состояние.

## Восстановление

Теперь симулируем сбой, принудительно выключив сервер.

```
=> CREATE DATABASE wal_checkpoint;
```

```
CREATE DATABASE
```

```
=> \c wal_checkpoint
```

```
You are now connected to database "wal_checkpoint" as user "student".
```

```
=> CREATE TABLE test(t text);
```

```
CREATE TABLE
```

```
=> INSERT INTO test VALUES ('Перед сбоем');
```

```
INSERT 0 1
```

```
student$ sudo head -n 1 /var/lib/postgresql/16/main/postmaster.pid
```

```
85667
```

```
student$ sudo kill -QUIT 85667
```

Сейчас на диске находятся журнальные записи, но табличные страницы не были сброшены на диск.

Проверим состояние кластера:

```
student$ sudo /usr/lib/postgresql/16/bin/pg_ctlcluster -D /var/lib/postgresql/16/main | grep state
```

```
Database cluster state:          in production
```

Состояние не изменилось. При запуске PostgreSQL поймет, что произошел сбой и требуется восстановление.

```
student$ sudo pg_ctlcluster 16 main start
```

```
student$ tail -n 6 /var/log/postgresql/postgresql-16-main.log
```

```
2025-03-06 18:04:47.461 MSK [86521] LOG:  redo starts at 0/1BA64F0
2025-03-06 18:04:47.491 MSK [86521] LOG:  invalid record length at 0/2004350: expected at
least 24, got 0
2025-03-06 18:04:47.491 MSK [86521] LOG:  redo done at 0/2004328 system usage: CPU: user:
0.00 s, system: 0.01 s, elapsed: 0.03 s
2025-03-06 18:04:47.510 MSK [86519] LOG:  checkpoint starting: end-of-recovery immediate
wait
2025-03-06 18:04:48.444 MSK [86519] LOG:  checkpoint complete: wrote 971 buffers (5.9%);
0 WAL file(s) added, 0 removed, 1 recycled; write=0.017 s, sync=0.905 s, total=0.937 s;
sync files=327, longest=0.017 s, average=0.003 s; distance=4471 kB, estimate=4471 kB;
lsn=0/2004350, redo lsn=0/2004350
2025-03-06 18:04:48.449 MSK [86518] LOG:  database system is ready to accept connections
```

```
student$ psql wal_checkpoint
```

```
=> SELECT * FROM test;
```

```
   t
-----
Перед сбоем
(1 row)
```

Как видим, таблица и данные восстановлены.

Теперь остановим экземпляр корректно. При такой остановке PostgreSQL выполняет контрольную точку, чтобы сбросить на диск все данные.

```
=> \q
```

```
student$ sudo pg_ctlcluster 16 main stop
```

Проверим состояние кластера:

```
student$ sudo /usr/lib/postgresql/16/bin/pg_ctlcluster -D /var/lib/postgresql/16/main | grep state
```

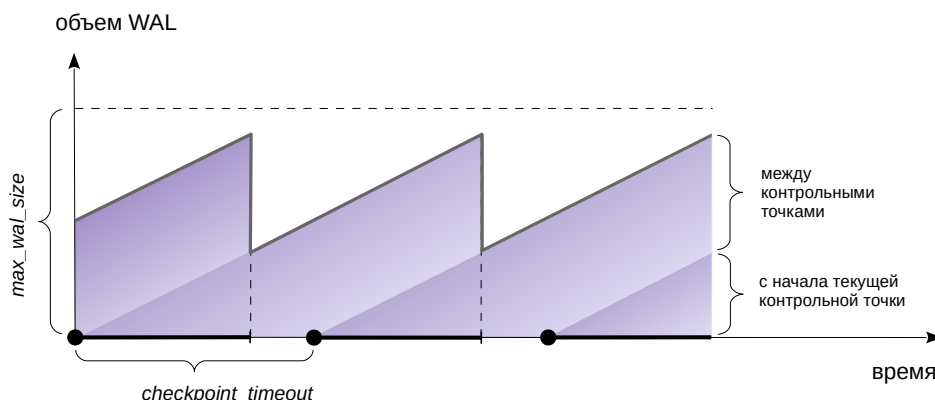
```
Database cluster state:          shut down
```

Теперь состояние — «shut down», что соответствует корректной остановке.

# Частота контрольных точек

`checkpoint_timeout` = 5min

`max_wal_size` = 1GB



12

Обычно контрольная точка настраивается из следующих соображений.

Сначала надо определиться, какая частота срабатываний нас устраивает (исходя из допустимого времени восстановления и объема журнальных файлов за это время при стандартной нагрузке). Чем реже можно позволить себе контрольные точки, тем лучше — это сокращает накладные расходы.

Устраивающая частота записывается в параметр `checkpoint_timeout` (значение по умолчанию — 5 минут — слишком мало, часто время увеличивают до получаса).

Однако возможна ситуация, когда нагрузка станет выше расчетной и за указанное время будет сгенерирован слишком большой объем журнальных записей. Для этого в параметре `max_wal_size` указывают общий допустимый объем журнальных записей.

Для восстановления после сбоя сервер должен хранить файлы с момента начала последней завершенной контрольной точки до начала текущей (объем между контрольными точками) плюс файлы, накопившиеся во время работы текущей контрольной точки. Поэтому общий объем можно оценить как

$(1 + \text{checkpoint\_completion\_target}) * \text{объем-между-контр-точками}$ .

Таким образом большая часть контрольных точек происходит по расписанию, раз в `checkpoint_timeout` единиц времени. Но при повышенной нагрузке контрольная точка вызывается чаще, чтобы постараться уложиться в объем `max_wal_size`.

## Сервер хранит журнальные файлы

необходимые для восстановления (обычно  $< max\_wal\_size$ )  
еще не прочитанные через слоты репликации  
еще не записанные в архив, если настроена непрерывная архивация  
не превышающие по объему  $wal\_keep\_size$   
не превышающие по объему  $min\_wal\_size$  (при переиспользовании)

## Настройки

$max\_wal\_size$	= 1GB
$wal\_keep\_size$	= 0
$wal\_recycle$	= on
$min\_wal\_size$	= 80MB

13

Надо понимать, что объем, указанный в параметре  $max\_wal\_size$ , может быть превышен. Это не жесткое ограничение, а ориентир для процесса checkpoint, влияющий на активность записи грязных буферов.

Кроме того, сервер не имеет права стереть журнальные файлы, еще не переданные через слоты репликации, и файлы, еще не записанные в архив при непрерывном архивировании. Это может привести к перерасходу места, так что если этот функционал используется, необходим постоянный мониторинг.

Можно на случай отставания реплики параметром  $wal\_keep\_size$  установить минимальный объем файлов журнала, остающихся после контрольной точки. Это не гарантирует, что журнальная запись сохранится до момента, когда она понадобится реплике, но все же позволяет работать без слота репликации.

По умолчанию журнальные файлы могут не удаляться, а просто переименовываться и использоваться заново. Параметр  $min\_wal\_size$  задает минимальный неудаляемый объем. Это позволяет сэкономить на постоянном создании и удалении файлов. Однако для файловых систем с сору-он-write быстрее создать новый файл, поэтому для них рекомендуется отключить переиспользование, установив  $wal\_recycle=off$ .

<https://postgrespro.ru/docs/postgresql/16/wal-configuration>

<https://postgrespro.ru/docs/postgresql/16/runtime-config-wal#RUNTIME-CONFIG-WAL-CHECKPOINTS>

## Объем журнала

Снова запустим экземпляр.

```
student$ sudo pg_ctlcluster 16 main start
```

Установим минимальное значение `min_wal_size` и отключим переиспользование, чтобы после контрольной точки оставалось не больше двух сегментов.

```
student$ psql wal_checkpoint
```

```
=> ALTER SYSTEM SET min_wal_size = '32MB';
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET wal_recycle = off;
```

```
ALTER SYSTEM
```

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Добавим строки в таблицу.

```
student$ psql wal_checkpoint
```

```
=> INSERT INTO test SELECT g.id::text FROM generate_series(1, 1_000_000) AS g(id);
```

```
INSERT 0 1000000
```

Список файлов журнала:

```
=> SELECT * FROM pg_ls_waldir() ORDER BY name;
```

name	size	modification
00000001000000000000000002	16777216	2025-03-06 18:04:56+03
00000001000000000000000003	16777216	2025-03-06 18:04:57+03
00000001000000000000000004	16777216	2025-03-06 18:04:57+03
00000001000000000000000005	16777216	2025-03-06 18:04:58+03

(4 rows)

Выполним ручную контрольную точку и опять посмотрим на журнал:

```
=> CHECKPOINT;
```

```
CHECKPOINT
```

```
=> SELECT * FROM pg_ls_waldir() ORDER BY name;
```

name	size	modification
00000001000000000000000005	16777216	2025-03-06 18:04:59+03
00000001000000000000000006	16777216	2025-03-06 18:04:59+03

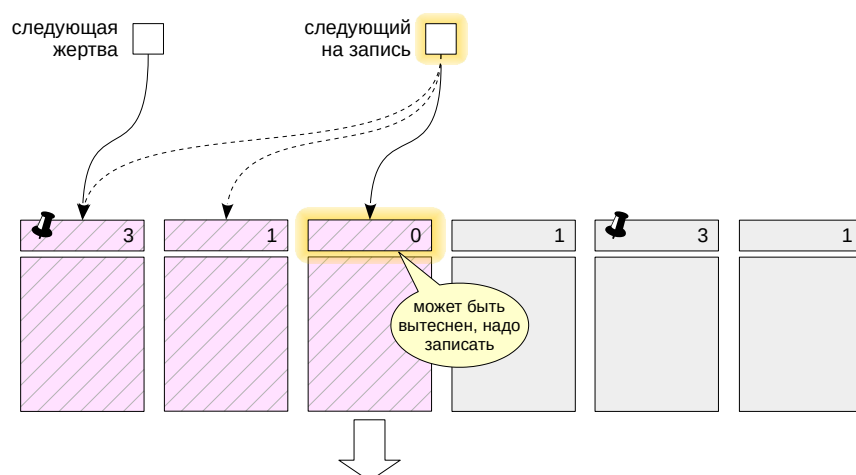
(2 rows)

После контрольной точки в журнале осталось не более двух сегментов, в том числе тот, который был текущим в момент ее начала. А если в кластере после начала контрольной точки происходили какие-либо изменения, в журнале могли появиться и другие сегменты.

Процесс фоновой записи

Настройка





Когда обслуживающий процесс собирается вытеснить страницу из буфера, он может обнаружить, что буфер грязный, и ему придется записывать эту страницу на диск. Чтобы снизить вероятность таких ситуаций, в дополнение к процессу контрольной точки (checkpoint) существует также процесс фоновой записи (background writer, bgwriter или просто writer).

Процесс фоновой записи использует тот же самый алгоритм поиска буферов для вытеснения, что и обслуживающие процессы, только использует свой указатель. Он может опережать указатель на «жертву», но никогда не отстает от него.

Записываются буферы, которые одновременно:

- содержат измененные данные (грязные),
- не закреплены (pin count = 0),
- имеют нулевое число обращений (usage count = 0).

Таким образом, фоновый процесс записи, «забегая вперед», находит те буферы, которые с большой вероятностью вскоре потребуются вытеснить. За счет этого обслуживающий процесс скорее всего обнаружит, что выбранный им буфер не является грязным.

## Алгоритм

уснуть на *bgwriter\_delay*

если в среднем за цикл запрашивается  $N$  буферов, то записать  
 $N * bgwriter_lru\_multiplier \leq bgwriter_lru\_maxpages$   
грязных буферов

## Настройки

<i>bgwriter_delay</i>	= 200ms
<i>bgwriter_lru_maxpages</i>	= 100
<i>bgwriter_lru_multiplier</i>	= 2.0
<i>bgwriter_flush_after</i>	= 512kB

Процесс фоновой записи работает циклами максимум по *bgwriter\_lru\_maxpages* страниц, засыпая между циклами на время *bgwriter\_delay*.

(Таким образом, если установить параметр *bgwriter\_lru\_maxpages* в ноль, процесс фактически не будет работать.)

Точное число буферов для записи определяется по среднему количеству буферов, которые запрашивались обслуживающими процессами с прошлого цикла (используется скользящее среднее, чтобы сгладить неравномерность между циклами, но при этом не зависеть от давней истории). Вычисленное количество буферов умножается на коэффициент *bgwriter\_lru\_multiplier*.

Если процесс совсем не обнаружил грязных буферов (то есть в системе ничего не происходит), он «впадает в спячку», из которой его выводит обращение серверного процесса за буфером. После этого процесс просыпается и опять работает обычным образом.

При необходимости записи большого количества данных сервер даёт указание ОС произвести промежуточный сброс данных в хранилище (fsync) по достижении объема, задаваемого параметром *bgwriter\_flush\_after*. Это уменьшает задержки фиксации транзакций и выполнение синхронизации в конце контрольной точки.

Процесс фоновой записи имеет смысл настраивать после того, как настроена контрольная точка. Совместно с контрольной точкой эти процессы должны успевать записывать грязные буферы до того, как они потребуются обслуживающим процессам.

## Контрольные точки

`checkpoint_warning = 30s`

`log_checkpoints = on`

статусы процессов `startup`, `checkpointer`

## Запись грязных буферов процессами

представления `pg_stat_bgwriter`, `pg_stat_io`

Параметр `checkpoint_warning` позволяет выводить в журнал предупреждение, если интервал между контрольными точками меньше, чем значение параметра. Если это происходит регулярно, следует подумать об увеличении `max_wal_size` или уменьшении интервала между контрольными точками `checkpoint_timeout`.

Включенный по умолчанию параметр `log_checkpoints` выводит в журнал подробную информацию о каждой выполненной контрольной точке.

Когда сервер находится в режиме восстановления, можно следить за статусом процессов `startup` и `checkpointer` средствами операционной системы, например с помощью утилиты `ps`.

Статистику работы процессов, записывающих грязные буферы (контрольной точки, фоновой записи и обслуживающих процессов), показывает представление `pg_stat_bgwriter`, а начиная с версии 16 — представление `pg_stat_io`.

В 17-ой версии PostgreSQL появится новое представление `pg_stat_checkpointer`, в которое из `pg_stat_bgwriter` будет перенесена информация о статистике процесса контрольной точки.

## Мониторинг

Параметр `checkpoint_warning` выводит предупреждение, если контрольные точки, вызванные переполнением размера журнальных файлов, выполняются слишком часто. Его значение по умолчанию:

```
=> SHOW checkpoint_warning;
```

```
checkpoint_warning
-----
30s
(1 row)
```

Его следует привести в соответствие со значением `checkpoint_timeout`.

Параметр `log_checkpoints` позволяет получать в журнале сообщений сервера информацию о выполняемых контрольных точках. По умолчанию (начиная с PostgreSQL 15) параметр включен:

```
=> SHOW log_checkpoints;
```

```
log_checkpoints
-----
on
(1 row)
```

Запишем что-нибудь в таблицу и выполним контрольную точку.

```
=> INSERT INTO test SELECT g.id::text FROM generate_series(1,100_000) AS g(id);
```

```
INSERT 0 100000
```

```
=> CHECKPOINT;
```

```
CHECKPOINT
```

Вот какую информацию можно узнать из журнала сообщений:

```
student$ tail -n 2 /var/log/postgresql/postgresql-16-main.log
```

```
2025-03-06 18:05:00.041 MSK [86780] LOG:  checkpoint starting: immediate force wait
2025-03-06 18:05:00.122 MSK [86780] LOG:  checkpoint complete: wrote 445 buffers (2.7%);
0 WAL file(s) added, 1 removed, 0 recycled; write=0.007 s, sync=0.050 s, total=0.082 s;
sync files=2, longest=0.049 s, average=0.025 s; distance=6274 kB, estimate=57043 kB;
lsn=0/63608E8, redo lsn=0/63608B0
```

Статистика работы процессов контрольной точки и фоновой записи отражается в одном общем представлении (раньше обе задачи решались одним процессом; затем их функции разделили, но представление осталось).

```
=> SELECT * FROM pg_stat_bgwriter \gx
```

```
-[ RECORD 1 ]-----+-----
checkpoints_timed    | 0
checkpoints_req      | 4
checkpoint_write_time | 71
checkpoint_sync_time | 1244
buffers_checkpoint   | 5842
buffers_clean        | 0
maxwritten_clean     | 0
buffers_backend      | 5179
buffers_backend_fsync | 0
buffers_alloc        | 6117
stats_reset          | 2025-03-06 18:04:47.44856+03
```

- `checkpoints_timed` — контрольные точки по расписанию (`checkpoint_timeout`);
- `checkpoints_req` — контрольные точки по требованию (`max_wal_size`) и выполненные вручную;
- `buffers_checkpoint` — страницы, сброшенные при контрольных точках;
- `buffers_clean` — страницы, сброшенные процессом фоновой записи;
- `buffers_backend` — страницы, сброшенные обслуживающими процессами;
- `maxwritten_clean` — количество остановок по достижению `bgwriter_lru_maxpages`.

В хорошо настроенной системе значение `buffers_backend` должно быть существенно меньше, чем сумма `buffers_checkpoint` и `buffers_clean`.

Большое значение `checkpoints_req` (по сравнению с `checkpoints_timed`) говорит о том, что контрольные точки происходят чаще, чем предполагалось.

Однако информация об объеме ввода-вывода, выдаваемая представлением `pg_stat_bgwriter`, не вполне корректна: в столбце `buffers_backend` отражены результаты работы не только клиентских процессов, но и некоторых других (например, автоочистки). В этот же столбец добавляются операции расширения файлов отношений, хотя это не запись из буферного кеша на диск.

Использованное нами ранее представление `pg_stat_io` предоставляет статистику в более информативном виде:

```
=> SELECT backend_type,
       sum(writes) AS writes,
       sum(fsyncs) AS fsyncs,
       sum(extends) AS extends,
       op_bytes
FROM pg_stat_io WHERE backend_type IN ('checkpointer', 'client backend', 'background writer')
GROUP BY backend_type, op_bytes;
```

backend_type	writes	fsyncs	extends	op_bytes
client backend	0	0	4428	8192
background writer	0	0		8192
checkpointer	5842	333		8192

(3 rows)

- `writes` — количество операций записи;
- `fsyncs` — количество вызовов `fsync`;
- `extends` — количество операций расширения отношений.

Процесс контрольной точки ограничивает размер хранимых журнальных файлов и сокращает время восстановления

Контрольная точка и фоновая запись сбрасывают на диск грязные буферы

Обслуживающие процессы могут сбрасывать грязные буферы, но не должны этим заниматься

1. Настройте выполнение контрольной точки раз в 30 секунд. Установите параметры *min\_wal\_size* и *max\_wal\_size* в 16 МБ.
2. Несколько минут с помощью утилиты *pgbench* подавайте нагрузку 100 транзакций/сек.
3. Измерьте, какой объем журнальных файлов был сгенерирован за это время.
4. Проверьте данные статистики и оцените, какой объем приходится в среднем на одну контрольную точку. Все ли контрольные точки выполнялись по расписанию? Как можно объяснить полученный результат?
5. Сбросьте настройки к значениям по умолчанию.

## 1. Настройка контрольной точки

```
=> ALTER SYSTEM SET checkpoint_timeout = '30s';
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET min_wal_size = '16MB';
```

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET max_wal_size = '16MB';
```

```
ALTER SYSTEM
```

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

## 2. Нагрузка

Инициализируем таблицы.

```
=> CREATE DATABASE wal_checkpoint;
```

```
CREATE DATABASE
```

```
student$ pgbench -i wal_checkpoint
```

```
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.20 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.42 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 0.25 s,
vacuum 0.07 s, primary keys 0.09 s).
```

Сбросим статистику.

```
=> SELECT pg_stat_reset_shared('bgwriter');
```

```
pg_stat_reset_shared
-----
(1 row)
```

Запускаем pgbench, предварительно запомнив позицию в журнале.

```
=> SELECT pg_current_wal_insert_lsn();
```

```
pg_current_wal_insert_lsn
-----
0/2C5D988
(1 row)
```

```
student$ pgbench -T 180 -R 100 wal_checkpoint
```

```
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 180 s
number of transactions actually processed: 18076
number of failed transactions: 0 (0.000%)
latency average = 56.811 ms
```



```
latency stddev = 195.010 ms
rate limit schedule lag: avg 52.151 (max 1542.151) ms
initial connection time = 4.151 ms
tps = 100.422710 (without initial connection time)
```

### 3. Объем журнальных файлов

```
=> SELECT pg_current_wal_insert_lsn();
```

```
pg_current_wal_insert_lsn
-----
0/810C3C8
(1 row)
```

```
=> SELECT pg_size_pretty('0/810C3C8'::pg_lsn - '0/2C5D988'::pg_lsn);
```

```
pg_size_pretty
-----
85 MB
(1 row)
```

### 4. Статистика

Число контрольных точек по расписанию и по требованию:

```
=> SELECT checkpoints_timed, checkpoints_req FROM pg_stat_bgwriter;
```

```
checkpoints_timed | checkpoints_req
-----+-----
1 | 6
(1 row)
```

Средний объем на контрольную точку:

```
=> SELECT pg_size_pretty(('0/810C3C8'::pg_lsn - '0/2C5D988'::pg_lsn)/7)
FROM pg_stat_bgwriter;
```

```
pg_size_pretty
-----
12 MB
(1 row)
```

Несмотря на то что в среднем объем журнальных записей за контрольную точку не превосходит установленного предела, часть контрольных точек выполнялась не по расписанию. Это говорит о неравномерности потока журнальных записей (вызванной выполнением автоочистки и другими причинами).

Поэтому в реальной системе замеры лучше выполнять на достаточно больших интервалах времени.

### 5. Настройки по умолчанию

```
=> ALTER SYSTEM RESET ALL;
```

```
ALTER SYSTEM
```

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```