

# Журналирование Журнал предзаписи



## 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.

Журнал предзаписи (WAL)

Логическое и физическое устройство журнала

Процесс упреждающей записи и восстановление

## Основная задача

возможность восстановления согласованности данных после сбоя

## Механизм

при изменении данных действие также записывается в журнал  
журнальная запись попадает на диск раньше измененных данных  
восстановление после сбоя — повторное выполнение потерянных операций с помощью журнальных записей

Основная причина существования журнала — необходимость восстановления согласованности данных в случае сбоя, при котором теряется содержимое оперативной памяти, в частности, буферного кеша. Тем самым обеспечивается выполнение свойства долговечности (буква «D» из набора свойств транзакций ACID).

Одновременно с изменением данных создается запись в журнале, содержащая достаточную информацию для повторения этой операции. Журнальная запись в обязательном порядке попадает на диск (или другое энергонезависимое устройство) до того, как туда попадет измененная страница — отсюда и название: «журнал предзаписи», «write-ahead log».

В случае сбоя можно прочесть журнал и при необходимости повторить те операции, которые уже были выполнены, но результат которых не успел попасть на диск.

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

## Изменение любых страниц в буферном кеше

в том числе страницы таблиц и индексов  
кроме нежурналируемых и временных таблиц

## Фиксация и отмена транзакций — буферы CLOG

## Файловые операции

создание и удаление файлов  
добавление и удаление страниц файлов  
создание и удаление каталогов

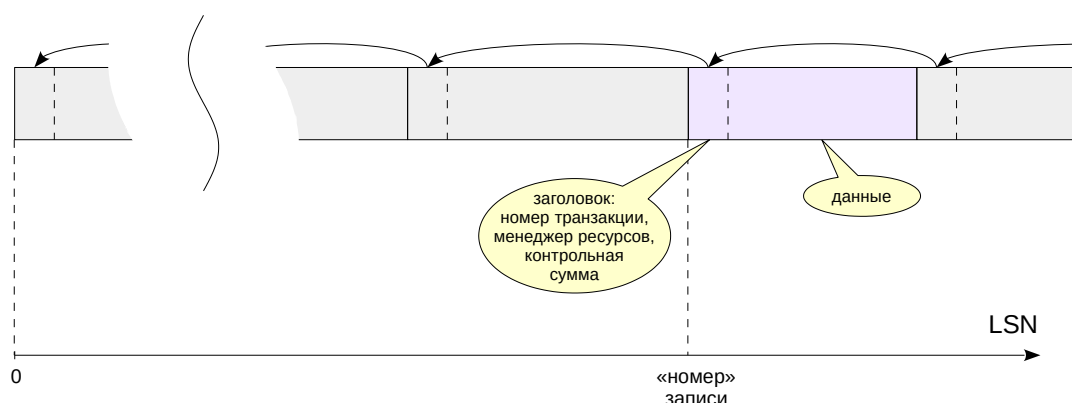
Журналировать нужно все операции, при выполнении которых возможна ситуация, что при сбое изменения (или часть изменений) не дойдут до диска.

В частности, в журнал записываются следующие действия:

- изменение страниц в буферном кеше (как правило, это страницы таблиц и индексов) — так как измененная страница попадает на диск не сразу;
- фиксация и отмена транзакций — точно так же, изменение статуса происходит в буфере CLOG и попадает на диск не сразу;
- файловые операции (создание и удаление файлов и каталогов, например, создание файлов при создании таблицы) — так как эти операции должны происходить синхронно с изменением данных.

При этом в журнал не записываются:

- операции с нежурналируемыми таблицами — их название говорит само за себя;
- операции с временными таблицами — они существуют не дольше, чем создавший их сеанс, и поэтому не нуждаются в восстановлении.



последовательность записей

«номер» записи — 64-битный LSN (log sequence number)

специальный тип `pg_lsn`

5

Логически журнал можно представить себе как последовательность записей различной длины. Каждая запись содержит данные о некоторой операции, предваренные заголовком. В заголовке, в числе прочего, указаны:

- номер транзакции, к которой относится запись;
- менеджер ресурсов — компонент системы, ответственный за данную запись, который понимает, как интерпретировать данные. Есть отдельные менеджеры для таблиц, для каждого типа индекса, для статуса транзакций и т. п.;
- контрольная сумма (CRC).

Начиная с версии 16 расширения могут создавать собственные менеджеры ресурсов, использующие специфический формат для своих записей WAL. Это поможет разработчикам создавать расширения, реализующие новые табличные и индексные методы доступа.

<https://postgrespro.ru/docs/postgresql/16/custom-rmgr>

Для того чтобы сослаться на определенную запись, используется тип данных `pg_lsn` (LSN = log sequence number) — 64-битное число, представляющее собой байтовое смещение до записи относительно начала журнала.

Начало каждой записи выравнивается по границе машинного слова, это одна из причин двоичной несовместимости WAL на разных платформах.

<https://postgrespro.ru/docs/postgresql/16/datatype-pg-lsn>

## Логическое устройство журнала

Список менеджеров ресурсов покажет утилита pg\_waldump:

```
student$ /usr/lib/postgresql/16/bin/pg_waldump -r list
```

```
XLOG
Transaction
Storage
CLOG
Database
Tablespace
MultiXact
RelMap
Standby
Heap2
Heap
Btree
Hash
Gin
Gist
Sequence
SPGist
BRIN
CommitTs
ReplicationOrigin
Generic
LogicalMessage
```

LSN выводится как два 32-битных числа в шестнадцатеричной системе через косую черту.

Текущая позиция в журнале:

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

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

При помощи утилиты pg\_waldump и появившегося в 15-й версии PostgreSQL расширения pg\_walinspect мы можем исследовать содержимое журнала предзаписи.

Создадим базу данных и установим расширение (чтобы воспользоваться расширением, роль должна быть включена в pg\_read\_server\_files или быть суперпользовательской):

```
=> CREATE DATABASE wal_log;
```

```
CREATE DATABASE
```

```
=> \c wal_log
```

You are now connected to database "wal\_log" as user "student".

```
=> CREATE EXTENSION pg_walinspect;
```

```
CREATE EXTENSION
```

Нам понадобится небольшая таблица:

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

```
CREATE TABLE
```

Получим текущую позицию в журнале, после чего вставим в таблицу строку:

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

```
pg_current_wal_insert_lsn
-----
0/1FD9FD0
(1 row)
```

```
=> INSERT INTO t VALUES ('FOO');
```

```
INSERT 0 1
```

Теперь позиция журнала такая:

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

```
pg_current_wal_insert_lsn
-----
0/1FDA050
(1 row)
```

Посмотрим, какие записи появились в журнале:

```
=> SELECT resource_manager, record_length, xid, start_lsn, prev_lsn, record_type, description
FROM pg_get_wal_records_info('0/1FD9FD0','0/1FDA050')
WHERE record_type in('INSERT+INIT','COMMIT')
ORDER BY start_lsn \gx
```

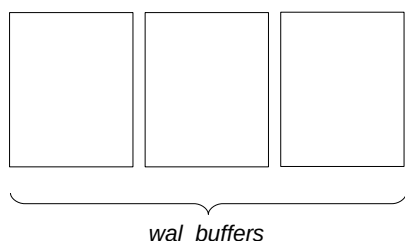
```
-[ RECORD 1 ]-----+-----
resource_manager | Heap
record_length    | 59
xid              | 746
start_lsn        | 0/1FD9FD0
prev_lsn         | 0/1FD9F90
record_type      | INSERT+INIT
description      | off: 1, flags: 0x00
-[ RECORD 2 ]-----+-----
resource_manager | Transaction
record_length    | 34
xid              | 746
start_lsn        | 0/1FDA028
prev_lsn         | 0/1FD9FD0
record_type      | COMMIT
description      | 2025-03-06 18:04:25.800424+03
```

Операция INSERT+INIT инициализирует страницу и добавляет в нее версию строки, операция COMMIT фиксирует транзакцию.

Для записи INSERT+INIT в журнале сохраняется блок данных, содержащий значения полей вставляемой версии строки. Расширение позволяет увидеть эти данные (в столбце block\_data можно различить коды символов добавленной строки):


```
=> SELECT * FROM pg_get_wal_block_info('0/1FD9FD0','0/1FDA028') \gx
```

```
-[ RECORD 1 ]-----+-----
start_lsn        | 0/1FD9FD0
end_lsn          | 0/1FDA028
prev_lsn         | 0/1FD9F90
block_id         | 0
reltablespace    | 1663
reldatabase      | 16390
relfilenode      | 16398
relforknumber    | 0
relblocknumber   | 0
xid              | 746
resource_manager | Heap
record_type      | INSERT+INIT
record_length    | 59
main_data_length | 3
block_data_length | 10
block_fpi_length | 0
block_fpi_info   | 
description      | off: 1, flags: 0x00
block_data       | \x01000208180009464f4f
block_fpi_data   |
```



В памяти

кольцевой буферный кеш

 `wal_buffers = -1`

1/32 shared\_buffers



На диске

файлы (сегменты) по 16 МБ

7

На диске журнал хранится в виде файлов (сегментов) в каталоге `PGDATA/pg_wal`. Каждый файл по умолчанию занимает 16 Мбайт. Размер сегмента может быть задан при инициализации кластера.

Журнальные записи попадают в текущий файл; когда он заполняется — начинает использоваться следующий.

В оперативной памяти для журнала выделены специальные буферы. Размер кеша задается параметром `wal_buffers` (значение по умолчанию подразумевает автоматическую настройку: выделяется 1/32 часть буферного кеша, но не меньше чем 64 Кбайт и не больше чем размер одного сегмента WAL).

Журнальный кеш устроен наподобие буферного кеша, но работает преимущественно в режиме кольцевого буфера: записи добавляются в «голову» буфера, а записываются на диск с «хвоста».



## Физическое устройство журнала

Размер кеша журнала в общей памяти:

```
=> SHOW wal_buffers;
```

```
wal_buffers
-----
4MB
(1 row)
```

Все журнальные файлы (сегменты) находятся в каталоге `/var/lib/postgresql/16/main/pg_wal/`, их также показывает специальная функция:

```
=> SELECT * FROM pg_ls_waldir() LIMIT 10;
```

```
      name      | size | modification
-----+-----+-----
000000010000000000000001 | 16777216 | 2025-03-06 18:04:26+03
(1 row)
```

Имена файлов составлены из трех чисел. Первое — номер линии времени (используется при восстановлении из архива), а два следующих — старшие разряды LSN.

Размер файлов можно задать при инициализации кластера, по умолчанию — 16 Мбайт.

Запись INSERT+INIT находится в этом файле:

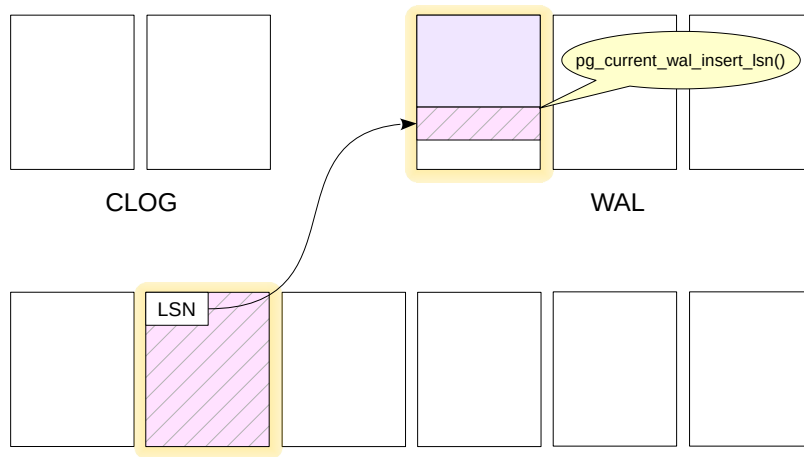
```
=> SELECT pg_walfile_name('0/1FDA050');
```

```
pg_walfile_name
-----
000000010000000000000001
(1 row)
```

Текущий сегмент:

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

```
pg_walfile_name
-----
000000010000000000000001
(1 row)
```

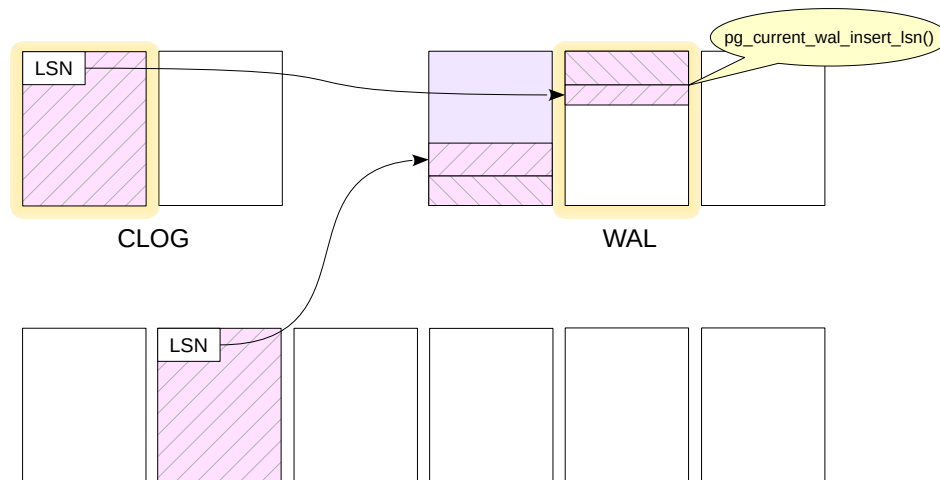


Проиллюстрируем сказанное выше про упреждающую запись. На слайде показаны три важные области общей памяти экземпляра:

- буферный кеш (размером `shared_buffers`),
- только что рассмотренный журнальный кеш WAL (размером `wal_buffers`),
- кеш состояния транзакций, называемый также CLOG (размером 128 страниц).

При изменении страницы данных в буферном кеше формируется журнальная запись. Она помещается в страницу журнала, а ссылка на запись (а точнее ее LSN + длина, то есть LSN следующей записи) записывается в специальное поле LSN в заголовке страницы данных.

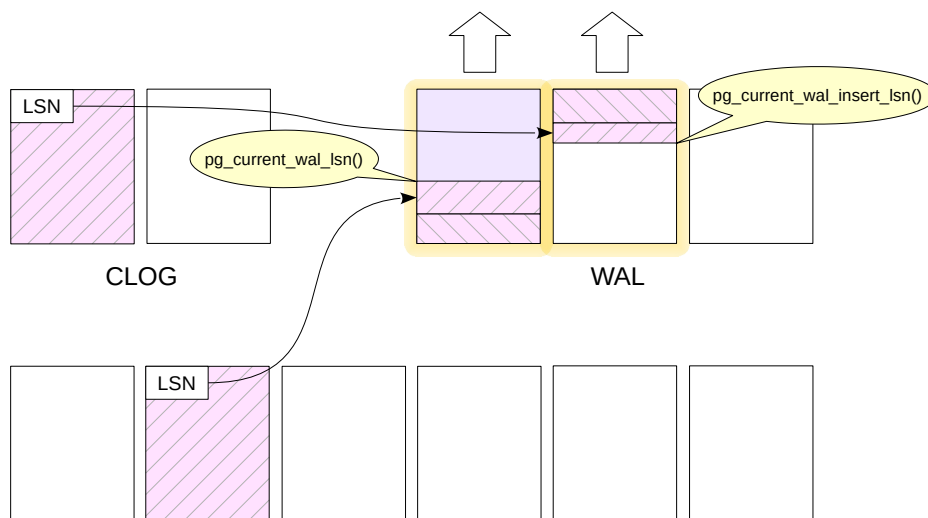
Позицию для записи можно узнать с помощью функции `pg_current_wal_insert_lsn()`.



Допустим, далее происходит фиксация транзакции. Для этого формируется еще одна журнальная запись, меняется бит состояния на странице CLOG и ссылка на эту запись проставляется в поле LSN измененной страницы.

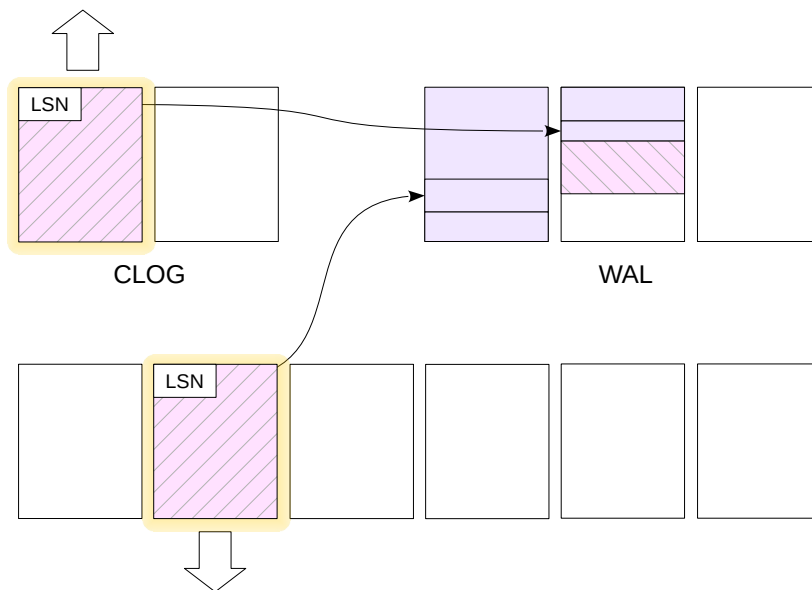
При вставке указатель `pg_current_wal_insert_lsn` сдвигается вперед.

Заметим, что между записями, относящимися к одной транзакции, могут попасть записи других транзакций, относящихся к любой БД. Журнал — общий для всего кластера.



Далее в какой-то момент (в какой именно — будет рассмотрено в теме «Настройка журнала») журнальные записи, которые еще не попали на диск, должны на него попасть.

Функция `pg_current_wal_lsn()` показывает последнюю запись, уже дошедшую до диска.



Только после того, как на диск попали журнальные записи, могут быть записаны и сами измененные страницы. Порядок контролируется с учетом LSN последнего изменения страницы и текущего состояния `pg_current_wal_lsn`. При этом работа продолжается, в журнал будут попадать новые и новые записи. Главное, чтобы запись с LSN последнего изменения страницы уже была на диске.

Если окажется, что страница данных должна быть записана (например, она вытесняется из буферного кеша), а журнальная запись еще не попала на диск, журнальные буферы сбрасываются принудительно.

## Упреждающая запись

Мы будем заглядывать в заголовок табличной страницы. Для этого понадобится расширение:

```
=> CREATE EXTENSION pageinspect;
```

```
CREATE EXTENSION
```

Начнем транзакцию.

```
=> BEGIN;
```

```
BEGIN
```

Текущая позиция и текущий сегмент журнала:

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

```
pg_current_wal_insert_lsn | pg_walfile_name
-----+-----
0/20529B0                | 000000010000000000000002
(1 row)
```

Изменим строку в таблице:

```
=> UPDATE t SET note = 'BAR';
```

```
UPDATE 1
```

Позиция в журнале изменилась:

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

```
pg_current_wal_insert_lsn
-----
0/20529F8
(1 row)
```

Этот же номер LSN (или меньший, если в журнал попали дополнительные записи) мы найдем и в заголовке измененной страницы:

```
=> SELECT lsn FROM page_header(get_raw_page('t',0));
```

```
lsn
-----
0/20529F8
(1 row)
```

Завершим транзакцию.

```
=> COMMIT;
```

```
COMMIT
```

Позиция в журнале снова изменилась:

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

```
pg_current_wal_insert_lsn
-----
0/2052A20
(1 row)
```

Размер фрагмента журнала (в байтах), соответствующий нашей транзакции:

```
=> SELECT pg_wal_lsn_diff('0/2052A20', '0/20529B0');
```

```
pg_wal_lsn_diff
-----
112
(1 row)
```

Безусловно, в журнал попадает информация обо всех действиях во всем кластере, но в данном случае мы рассчитываем на то, что в системе ничего не происходит.

Вот сами записи:

```
=> SELECT xid, record_type, record_length, end_lsn-start_lsn occupied
FROM pg_get_wal_records_info('0/20529B0', '0/2052A20')
ORDER BY start_lsn;
```

xid	record_type	record_length	occupied
788	HOT_UPDATE	70	72
788	COMMIT	34	40

(2 rows)

Заметим, что из-за выравнивания длина записи обычно меньше, чем занимаемое ей место (столбец occupied).

Увидеть содержимое журнала можно и с помощью утилиты `pg_waldump`. Она может работать с диапазоном LSN (как в этом примере), выбрать записи для определенного отношения и отдельного слоя, а также для отдельной страницы или указанной транзакции. В отличие от расширения `pg_walinspect`, утилитой можно пользоваться и на остановленном сервере, при этом ей нужен доступ на чтение журнальных файлов, поэтому мы выполним команду от имени суперпользователя:

```
student$ sudo /usr/lib/postgresql/16/bin/pg_waldump \
-p /var/lib/postgresql/16/main/pg_wal \
-s 0/20529B0 -e 0/2052A20 \
0000000100000000000000002 000000010000000000000002 \
| fold -sw 100

rmgr: Heap len (rec/tot): 70/ 70, tx: 788, lsn: 0/020529B0, prev 0/02052970,
desc: HOT_UPDATE old_xmax: 788, old_off: 1, old_infobits: [], flags: 0x01, new_xmax: 0, new_off: 2,
blkref #0: rel 1663/16390/16398 blk 0
rmgr: Transaction len (rec/tot): 34/ 34, tx: 788, lsn: 0/020529F8, prev 0/020529B0,
desc: COMMIT 2025-03-06 18:04:28.245992 MSK
```

Мы видим заголовки журнальных записей:

- операция `HOT_UPDATE`, относящаяся к измененной странице (rel+blk),
- операция `COMMIT` с указанием времени.

## Алгоритм (упрощенный)

при старте сервера после сбоя  
(состояние кластера в `pg_control` отличается от «shut down»):

1. для каждой журнальной записи:
  - 1.1. определить страницу, к которой относится эта запись
  - 1.2. применить запись, если ее LSN больше, чем LSN страницы
2. перезаписать нежурналируемые таблицы init-файлами

Если в работе сервера произошел сбой, то при последующем запуске процесс `startup` (запускаемый `postmaster`-ом в самом начале работы) обнаружит это, посмотрев в файл `pg_control` и увидев статус, отличный от «shut down». Тогда автоматически будет выполнено восстановление.

Процесс `startup` будет последовательно читать журнал и применять 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 — для этого `postmaster` запускает необходимые фоновые процессы.

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

В конце процесса все нежурналируемые таблицы перезаписываются с помощью образов в init-файлах.

Процесс восстановления можно ускорить, включив реализованную для этого в PostgreSQL 15 предвыборку данных журнала. Эта предвыборка включается параметром `recovery_prefetch`. В этом случае значение параметра `wal_decode_buffer_size` определяет, как далеко сервер будет заглядывать в журнал, чтобы найти там номера нужных страниц.

Приведенный алгоритм является упрощенным. В частности, мы ничего не сказали о том, с какого места надо начинать чтение журнальных записей (это будет рассмотрено в теме «Контрольная точка»).



Использование буферов в оперативной памяти приводит к необходимости журналирования

Журнал содержит информацию, позволяющую повторно выполнить операции после сбоя и восстановить согласованность

Журнал всегда записывается на диск до того, как записываются измененные страницы данных

1. Создайте таблицу с первичным ключом и добавьте в нее несколько строк. Сколько байт занимают сгенерированные журнальные записи?
2. Чем можно объяснить довольно большое их число? Просмотрите заголовки этих журнальных записей и проверьте свои предположения.
3. Измените добавленные в таблицу строки. Снова измените строки, но не фиксируйте транзакцию. Сымитируйте сбой, прервав процесс postmaster.  
Запустите сервер и убедитесь, что зафиксированные изменения не пропали, а незафиксированная транзакция оборвана. Найдите информацию о восстановлении после сбоя в журнале сообщений сервера.

16

2. Можно воспользоваться утилитой `pg_waldump` или расширением `pg_walinspect`.

3. Воспользуйтесь командой

```
$ sudo kill -QUIT номер-процесса
```

Номер процесса находится в первой строке файла `postmaster.pid` в каталоге `PGDATA` сервера.

## 1. Размер журнальных записей

```
=> CREATE DATABASE wal_log;
```

```
CREATE DATABASE
```

```
=> \c wal_log
```

You are now connected to database "wal\_log" as user "student".

Запомним начальную позицию в журнале:

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

```
pg_current_wal_insert_lsn
-----
0/1FD1AC8
(1 row)
```

Создадим таблицу и добавим строки:

```
=> CREATE TABLE t(
  id integer PRIMARY KEY,
  s text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO t VALUES (1, 'A'), (2, 'B'), (3, 'C');
```

```
INSERT 0 3
```

Запомним конечную позицию:

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

```
pg_current_wal_insert_lsn
-----
0/1FD5380
(1 row)
```

Размер журнальных записей:

```
=> SELECT '0/1FD5380'::pg_lsn - '0/1FD1AC8'::pg_lsn;
```

```
?column?
-----
14520
(1 row)
```

## 2. Состав журнальных записей

Журнальный файл:

```
=> SELECT pg_walfile_name('0/1FD1AC8');
```

```
pg_walfile_name
-----
000000010000000000000001
(1 row)
```

Понадобится расширение pg\_walinspect:

```
=> CREATE EXTENSION pg_walinspect;
```

```
CREATE EXTENSION
```

Смотрим записи:

```
=> SELECT start_lsn, xid, resource_manager, record_type,
  regexp_match(block_ref, 'rel (\d+/\d+/\d+) fork (\w+) blk (\w+)') AS bref
FROM pg_get_wal_records_info('0/1FD1AC8', '0/1FD5380');
```

start_lsn	xid	resource_manager	record_type	bref
0/1FD1AC8	0	Storage	CREATE	

0/1FD1AF8	744	Heap	INSERT	{1663/16390/1247,main,14}
0/1FD1BD0	744	Btree	INSERT_LEAF	{1663/16390/2703,main,2}
0/1FD1C10	744	Btree	INSERT_LEAF	{1663/16390/2704,main,2}
0/1FD1C50	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,2}
0/1FD1CA8	744	Btree	INSERT_LEAF	{1663/16390/2673,main,4}
0/1FD1CF0	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD1D38	744	Heap	INSERT	{1663/16390/1247,main,14}
0/1FD1E10	744	Btree	INSERT_LEAF	{1663/16390/2703,main,2}
0/1FD1E50	744	Btree	INSERT_LEAF	{1663/16390/2704,main,4}
0/1FD1E90	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,3}
0/1FD1EE8	744	Btree	INSERT_LEAF	{1663/16390/2673,main,4}
0/1FD1F30	744	Btree	INSERT_LEAF	{1663/16390/2674,main,5}
0/1FD1F78	744	Heap	INSERT	{1663/16390/1259,main,0}
0/1FD2060	744	Btree	INSERT_LEAF	{1663/16390/2662,main,2}
0/1FD20A0	744	Btree	INSERT_LEAF	{1663/16390/2663,main,2}
0/1FD20E0	744	Btree	INSERT_LEAF	{1663/16390/3455,main,1}
0/1FD2120	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,17}
0/1FD2250	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2290	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD22D0	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2310	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2350	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,17}
0/1FD2678	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD26C0	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2700	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2748	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2788	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD27D0	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2810	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2858	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2898	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD28E0	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2920	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2968	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD29A8	744	Heap	INSERT	{1664/0/1214,main,0}
0/1FD29F8	744	Btree	INSERT_LEAF	{1664/0/1232,main,1}
0/1FD2A40	744	Btree	INSERT_LEAF	{1664/0/1233,main,1}
0/1FD2A80	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,3}
0/1FD2AD8	744	Btree	INSERT_LEAF	{1663/16390/2673,main,8}
0/1FD2B20	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD2B68	744	Standby	LOCK	
0/1FD2B98	744	Storage	CREATE	
0/1FD2BC8	744	Heap	INSERT	{1663/16390/1259,main,0}
0/1FD2C98	744	Btree	INSERT_LEAF	{1663/16390/2662,main,2}
0/1FD2CD8	744	Btree	INSERT_LEAF	{1663/16390/2663,main,2}
0/1FD2D28	744	Btree	INSERT_LEAF	{1663/16390/3455,main,1}
0/1FD2D68	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,17}
0/1FD2F18	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2F60	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD2FA0	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD2FE8	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3028	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD3070	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD30B0	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,17}
0/1FD3358	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,56}
0/1FD3408	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD3450	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3490	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD34D8	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3518	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD3560	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD35A0	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD35E8	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3628	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD3670	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD36B0	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD36F8	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3738	744	Storage	CREATE	
0/1FD3768	744	Standby	LOCK	
0/1FD3798	744	Heap	INSERT	{1663/16390/1259,main,0}
0/1FD3868	744	Btree	INSERT_LEAF	{1663/16390/2662,main,2}
0/1FD38A8	744	Btree	INSERT_LEAF	{1663/16390/2663,main,2}
0/1FD3900	744	Btree	INSERT_LEAF	{1663/16390/3455,main,1}
0/1FD3940	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,56}
0/1FD3A70	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD3AB8	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3AF8	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD3B40	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD3B80	744	Heap	INSERT	{1663/16390/2610,main,0}
0/1FD3C50	744	Btree	INSERT_LEAF	{1663/16390/2678,main,1}

0/1FD3C90	744	Btree	INSERT_LEAF	{1663/16390/2679,main,1}
0/1FD3CD0	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,3}
0/1FD3D50	744	Btree	INSERT_LEAF	{1663/16390/2673,main,8}
0/1FD3D98	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD3DE0	744	Btree	INSERT_LEAF	{1663/16390/2673,main,8}
0/1FD3E28	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD3E70	744	XLOG	FPI	{1663/16390/16395,main,0}
0/1FD3F00	744	Heap	INPLACE	{1663/16390/1259,main,0}
0/1FD3FC0	744	Heap	INPLACE	{1663/16390/1259,main,0}
0/1FD4098	744	Heap	HOT_UPDATE	{1663/16390/1259,main,0}
0/1FD40E8	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,3}
0/1FD4140	744	Btree	INSERT_LEAF	{1663/16390/2673,main,8}
0/1FD4188	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD41D0	744	Storage	CREATE	
0/1FD4200	744	Standby	LOCK	
0/1FD4230	744	Heap	INSERT	{1663/16390/1259,main,0}
0/1FD4300	744	Btree	INSERT_LEAF	{1663/16390/2662,main,2}
0/1FD4340	744	Btree	INSERT_LEAF	{1663/16390/2663,main,2}
0/1FD4388	744	Btree	INSERT_LEAF	{1663/16390/3455,main,1}
0/1FD43C8	744	Heap2	MULTI_INSERT	{1663/16390/1249,main,56}
0/1FD4478	744	Btree	INSERT_LEAF	{1663/16390/2658,main,15}
0/1FD44B8	744	Btree	INSERT_LEAF	{1663/16390/2659,main,10}
0/1FD44F8	744	Heap	INSERT	{1663/16390/2610,main,0}
0/1FD45C0	744	Btree	INSERT_LEAF	{1663/16390/2678,main,1}
0/1FD4600	744	Btree	INSERT_LEAF	{1663/16390/2679,main,1}
0/1FD4640	744	Heap	INSERT	{1663/16390/2606,main,2}
0/1FD4700	744	Btree	INSERT_LEAF	{1663/16390/2579,main,1}
0/1FD4740	744	Btree	INSERT_LEAF	{1663/16390/2664,main,1}
0/1FD4788	744	Btree	INSERT_LEAF	{1663/16390/2665,main,1}
0/1FD47D0	744	Btree	INSERT_LEAF	{1663/16390/2666,main,1}
0/1FD4810	744	Btree	INSERT_LEAF	{1663/16390/2667,main,1}
0/1FD4850	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,3}
0/1FD48A8	744	Btree	INSERT_LEAF	{1663/16390/2673,main,5}
0/1FD48F0	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD4938	744	Heap2	MULTI_INSERT	{1663/16390/2608,main,3}
0/1FD4990	744	Btree	INSERT_LEAF	{1663/16390/2673,main,8}
0/1FD49D8	744	Btree	INSERT_LEAF	{1663/16390/2674,main,7}
0/1FD4A20	744	XLOG	FPI	{1663/16390/16396,main,0}
0/1FD4AB0	744	Heap	INPLACE	{1663/16390/1259,main,0}
0/1FD4B70	744	Heap	INPLACE	{1663/16390/1259,main,0}
0/1FD4C30	744	Transaction	COMMIT	
0/1FD5178	745	Heap	INSERT+INIT	{1663/16390/16391,main,0}
0/1FD51B8	745	Btree	NEWROOT	{1663/16390/16396,main,1}
0/1FD5218	745	Btree	INSERT_LEAF	{1663/16390/16396,main,1}
0/1FD5258	745	Heap	INSERT	{1663/16390/16391,main,0}
0/1FD5298	745	Btree	INSERT_LEAF	{1663/16390/16396,main,1}
0/1FD52D8	745	Heap	INSERT	{1663/16390/16391,main,0}
0/1FD5318	745	Btree	INSERT_LEAF	{1663/16390/16396,main,1}
0/1FD5358	745	Transaction	COMMIT	

(130 rows)

Вначале (до первой операции COMMIT) происходит активная работа с таблицами и индексами системного каталога. За счет этого размер записей и получился существенно больше, чем в демонстрации.

### 3. Восстановление после сбоя

Обновляем строки:

```
=> UPDATE t SET s = 'FOO';
```

```
UPDATE 3
```

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE t SET s = 'BAR'; -- не фиксируем транзакцию
```

```
UPDATE 3
```

Прерываем основной серверный процесс.

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

```
117043
```

```
student$ sudo kill -QUIT 117043
```

```
student$ sudo pg_ctlcluster 16 main status
```

```
pg_ctl: no server running
```

Запускаем сервер.

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

Проверяем изменения:

```
student$ psql wal_log
```

```
=> SELECT * FROM t;
```

```
id | s
----+-----
 1 | F00
 2 | F00
 3 | F00
(3 rows)
```

Журнал сообщений:

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

```
2025-03-06 18:16:41.358 MSK [118213] LOG:  redo starts at 0/1BA64F0
2025-03-06 18:16:41.378 MSK [118213] LOG:  invalid record length at 0/1FDAF18: expected
at least 24, got 0
2025-03-06 18:16:41.378 MSK [118213] LOG:  redo done at 0/1FDAEF0 system usage: CPU:
user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
2025-03-06 18:16:41.548 MSK [118211] LOG:  checkpoint starting: end-of-recovery immediate
wait
2025-03-06 18:16:42.090 MSK [118211] LOG:  checkpoint complete: wrote 939 buffers (5.7%);
0 WAL file(s) added, 0 removed, 0 recycled; write=0.012 s, sync=0.524 s, total=0.545 s;
sync files=308, longest=0.009 s, average=0.002 s; distance=4306 kB, estimate=4306 kB;
lsn=0/1FDAF18, redo lsn=0/1FDAF18
2025-03-06 18:16:42.097 MSK [118210] LOG:  database system is ready to accept connections
```