Записки программиста

Блог о программировании, а также электронике, радио и 3D печати

Внутренности PostgreSQL: журнал предзаписи (WAL)

23 января 2023

PostgreSQL хранит данные в страницах, а страницы кэшируются в разделяемых буферах. Казалось бы, в случае аварийной остановки грязные страницы не будут записаны на диск, и часть данных пропадет. Чтобы такого не происходило, СУБД пишет журнал предзаписи, он же Write Ahead Log, или WAL.

Теория

Идея заключается в следующем. Перед тем, как изменить что-либо на странице, СУБД делает соответствующую запись в WAL (он же XLOG, от Transaction Log). WAL хранится на диске. Запись содержит что-то уровня «открыть такую-то страницу и записать N таких-то байт по такому-то смещению». В случае аварийной остановки при следующем запуске СУБД откроет WAL и проиграет все записи из него. То есть, сделает все измения, которые могли потеряться.

Смещение записи относительно начала журнала называется Log Sequence Number, или LSN. Записи в среднем довольно короткие, десятки байт, и писать их по одной невыгодно. Поэтому в разделяемой памяти есть кольцевой буфер для WAL-записей. Размер буфера определяется параметром wal_buffers и по умолчанию составляет 1/32 от размера разделяемых буферов, но не менее 64 Кб и не более 16 Мб.

Как правило, страница не может быть вытеснена раньше WAL-записи о последних изменениях на этой странице. Это необходимо для корректного восстановления системы после сбоя. Ведь WAL-записи не могут применяться к страницам из будущего. По этой причине в PageHeaderData в поле pd_lsn хранится LSN. Исключением из правила являются нежурналируемые таблицы.

Во время восстановления WAL-запись должна быть пропущена, если на диске записана страница с большим LSN. Эта страница была вытеснена на диск после того, как была сделана текущая запись в WAL. Страница содержит более свежие данные. Далее в WAL для нее могут быть другие записи, с большим LSN.

Примечание: В PostgreSQL пользователю доступен тип pg_lsn, а также функции pg_current_wal_lsn(), pg_walfile_name(), pg_ls_waldir() и прочие. Подробности ищите в документации.

Запись в WAL происходит с fsync(). Кроме того, записи имеют контрольные суммы. При таком подходе данные не могут быть потеряны. Конечно, если только диск и/или сервер целиком не выйдут из строя. Но это при любом раскладе будет означать восстановление из резервной копии. Заметьте, что работа с WAL — это последовательная запись на диск, что является более дешевой операцией по сравнению со случайной записью в кучу.

По умолчанию в PostgreSQL транзакция считается завершенной успешно, когда все соответствующие ей WAL-записи были записаны на диск с fsync(). Установив `synchronous_commit = off` можно выжать больше производительности в обмен на потенциальную потерю нескольких последних транзакций в случае сбоя. Подробности описаны в документации.

Если СУБД будет работать долго, она напишет большой WAL, что приведет к медленному восстановлению в случае сбоя. Эта проблема решается при помощи контрольных точек, или checkpoints. Успешный checkpoint означает, что состояние всех страниц на такой-то момент времени (точнее, на такой-то LSN) было записано на диск. В случае восстановления после сбоя СУБД достаточно проиграть записи WAL только с последнего checkpoint'а.

Контрольную точку можно создать вручную, выполнив команду `CHECKPOINT`. Также чекпоинты периодически создает фоновый процесс под названием checkpointer. Его настройки подробно описаны в официальной документации.

Когда страница изменяется первый раз после контрольной точки, в WAL пишется полная копия страницы. Это необходимо по той причине, что запись грязной страницы в кучу может быть прервана. Страница окажется испорчена и мы не сможем докатить на нее изменения из WAL. Наличие копии страницы в WAL устраняет проблему. Это поведение можно изменить, указав в postgresql.conf параметр `full_page_writes = off`. Разумеется, настроенная таким образом система может терять данные.

В отличие от некоторых других СУБД, пишущих в WAL как undo, так и redo записи, PostgreSQL пишет только redo. То есть, какие действия нужно совершить, чтобы докатить изменения, но не откатить. Данное решение имеет свои сильные и слабые стороны.

К сильным сторонам относится уменьшение размера WAL, а также существенное упрощение алгоритма восстановления после сбоев (факт проигрывания undo записей при восстановлении нужно логировать, в отличие от redo). Как результат, упрощается тестирование этого алгоритма, в том числе при изменениях между версиями PostgreSQL, в случаях когда СУБД падает во время процесса восстановления, и так далее. Исполнение долгих транзакций не может быть прервано по причине удаления старых undo записей, как это бывает в других системах.

Недостатком является тот факт, что в куче находятся и старые, и новые кортежи. Для удаления старых кортежей нужно периодически делать VACUUM. В СУБД с undo записями старые кортежи вытесняются из кучи в undo записи. VACUUM в таких СУБД не нужен.

Также, в отличие от других СУБД, PostgreSQL берет информацию об успешном checkpoint не из в WAL, а из отдельного файла pg_control. Файл содержит менее 512-и байт данных и имеет контрольную сумму. Это является слабым местом, поскольку запись в такой файл не обязана быть атомарной. Тем не менее, на практике все работает более-менее нормально. Отчасти, потому что грамотные DBA знают про такую особенность и кладут pg_control на журналируемую ФС, а простые DBA держат *вообще все* на ext4 с включенным журналированием.

Вместо использования отдельного файла более правильно было бы проигрывать WAL от конца к началу в поисках последнего checkpoint.
На данный момент это не реализовано в PostgreSQL.

Если СУБД падает во время записи в WAL, то последняя запись может быть записана частично, и контрольная сумма для нее не сойдется. По этой причине некоторые СУБД игнорируют последнюю запись, если она повреждена. Однако повреждение WAL также может быть признаком деградации жесткого диска, или bit rot. Такие СУБД имеют больше шансов потерять данные незаметно для пользователя. Если PostgreSQL видит

битую запись в WAL, он отказывается стартовать, и требует ручного вмешательства DBA. Заинтересованным читателям предлагается проверить это самостоятельно. Radare2 отлично подойдет для того, чтобы подправить пару байт в WAL.

Практика

Создадим новую базу данных с одной-единственной таблицей:

```
CREATE TABLE phonebook(
  "name" VARCHAR(64) NOT NULL,
  "phone" INT NOT NULL);
```

Таблица очень простая, без каких-либо индексов и без TOAST. В каталоге pg_wal при этом должен быть один файл размером 16 Мб:

```
$ ls -lah /home/eax/projects/pginstall/data-master/pg_wal
total 17M
drwx------  3 eax eax 4.0K Jan 08 14:08 .
drwx------ 19 eax eax 4.0K Jan 08 14:08 ..
-rw-------  1 eax eax  16M Jan 08 14:10 000000010000000000000001
drwx------  2 eax eax 4.0K Jan 08 14:08 archive_status
```

Также как и с кучей, это файл называется сегментом и разбит на страницы размером по 8 Кб. Размер сегмента может быть переопределен при помощи флага компиляции `--wal-segsize`, а размер страницы в WAL — при помощи `--with-wal-blocksize`. Номера сегментов строго возрастают. Никакого переиспользования (wraparound) номеров не предусмотрено, поскольку номеров много. Пройдет очень много времени прежде, чем они закончатся.

Для чтения сегментов предусмотрена утилита pg_waldump:

```
$ pg_waldump -p ~/projects/pginstall/data-master/pg_wal \
    000000010000000000000001 | wc -l

pg_waldump: error: error in WAL record at 0/1B144C0: invalid record
length at 0/1B144F8: wanted 24, got 0

22002
```

На сообщение об ошибке не обращайте внимания. Так утилита говорит о том, что нашла конец журнала.

Видим, что в журнале немало записей. Они там появились после инициализации базы данных. Также создание таблицы phonebook приводит к изменению таблиц каталога, что создает больше одной записи в журнале.

Допишем немного данных в таблицу:

```
INSERT INTO phonebook ("name", "phone")
VALUES ('Alice', 123), ('Bob', 456), ('Charlie', 789);
```

Повторив предыдущую команду с pg_waldump видим, что в журнале теперь 22007 записей. Воспользовавшись `tail -n 5`, мы можем узнать, что конкретно записал наш INSERT:

```
rmgr: Heap        len (rec/tot): 67/67, tx:        739,   ⏎
    lsn: 0/01B144F8, prev 0/01B144C0, desc: INSERT+INI    ⏎
    off 1 flags 0x08, blkref #0: rel 1663/16384/16389 blk 0
rmgr: Heap        len (rec/tot): 63/63, tx:        739,   ⏎
    lsn: 0/01B14540, prev 0/01B144F8, desc: INSERT        ⏎
    off 2 flags 0x08, blkref #0: rel 1663/16384/16389 blk 0
rmgr: Heap        len (rec/tot): 67/67, tx:        739,   ⏎
    lsn: 0/01B14580, prev 0/01B14540, desc: INSERT        ⏎
    off 3 flags 0x08, blkref #0: rel 1663/16384/16389 blk 0
rmgr: Transaction len (rec/tot): 46/46, tx:        739,   ⏎
    lsn: 0/01B145C8, prev 0/01B14580, desc: COMMIT        ⏎
    2023-01-08 14:14:27.585865 MSK
rmgr: Standby     len (rec/tot): 50/50, tx:          0,   ⏎
    lsn: 0/01B145F8, prev 0/01B145C8, desc: RUNNING_XACTS ⏎
    nextXid 740 latestCompletedXid 739 oldestRunningXid 740
```

Смысл полей rmgr, len и прочих станет понятен ниже, когда мы рассмотрим структуру XLogRecord.

Что же до файла pg_control, он имеет путь $PGDATA/global/pg_control. Для его чтения есть утилита pg_controldata. Убедиться, что файл содержит информацию о последнем checkpoint'е можно так:

```
pg_controldata ~/projects/pginstall/data-master/ > before.txt
psql -c 'CHECKPOINT;'
pg_controldata ~/projects/pginstall/data-master/ > after.txt
diff before.txt after.txt
```

Или даже так:

```
pg_controldata ~/projects/pginstall/data-master/ | \
  grep 'Latest checkpoint'
```

Страницы в WAL начинаются с заголовка XLogPageHeaderData размером 20 байт. Если проставлен флаг XLP_LONG_HEADER, заголовок имеет размер 36 байт и тип XLogLongPageHeaderData. XLogLongPageHeaderData включает в себя XLogPageHeaderData, а также избыточные данные для сверки с pg_control. К избыточным данным относятся 64-х битный идентификатор системы, а также размер сегментов и страниц WAL. Длинная версия заголовка пишется в первую страницу сегмента. Описание этих структур находится в файле xlog_internal.h.

Каждая запись в WAL начинается с заголовка фиксированного размера 24 байта:

```
typedef struct XLogRecord
{
  uint32        xl_tot_len;  /* общая длина записи */
  TransactionId xl_xid;      /* ID транзакции, создавшей запись */
  XLogRecPtr    xl_prev;     /* указатель на предыдущую запись */
  uint8         xl_info;     /* флаги и 3 бита id операции */
  RmgrId        xl_rmid;     /* resource manager этой записи */

  /* здесь 2 байта 0x00 для выравнивания */

  pg_crc32c     xl_crc;      /* контрольная сумма записи */
} XLogRecord;
```

Следом идут данные, которые зависят от типа записи. Подробности ищите в файле xlogrecord.h.

За обработку записей конкретного типа отвечают так называемые resource managers. RmgrId кодируется одним байтом. На момент написания этих строк существует 22 разных resource managers, среди которых Heap, Btree, Generic, и другие. Каждый менеджер должен уметь проигрывать свои записи (ищите функции heap_redo, btree_redo, …), декодировать их в текстовое описание (heap_desc, btree_desc, …), и так далее. Подробности смотрите в rmgrlist.h.

Характерно, что для кучи существует два resource managers — Heap и Heap2. Так получилось по той причине, что трех бит (XLOG_HEAP_OPMASK) не хватило для кодирования всех требуемых операций. Поэтому resource manager для кучи распался на два, по 8 операций на каждый. Наверное, можно было бы изменить формат XLogRecord, но это наверняка сломает сторонние инструменты для резервного копирования и всякого такого.

Содержимое pg_control кодируется структурой ControlFileData. Ее описание находится в pg_control.h. В файлах pg_waldump.c и pg_controldata.c можно ознакомиться с исходниками одноименных утилит. Кроме того, есть расширение pg_walinspect. Оно аналогично pageinspect, только для WAL. Его исходники ищите в каталоге contrib/pg_walinspect.

Компонент, отвечающий за работу с WAL и pg_control, называется WAL manager. Его реализация находится в xlog.c, xlog.h и сопутствующих файлах. Наибольший интерес представляют функции XLogInsertRecord(), XLogFlush(), а также функция CreateCheckPoint(). Код инициализации находится в функциях XLOGShmemSize(), XLOGShmemInit() и StartupXLOG().

Как и buffer manager, WAL manager не является каким-то выделенным процессом. Однако есть выделенный процесс walwriter, занимающийся записью WAL. Он очень похож на ранее рассмотренный bgwriter, только вызывает в цикле XLogBackgroundFlush() из xlog.c. Полная реализация находится в walwriter.c.

Заключение

Данная заметка не претендует на исчерпывающее описание всех тонкостей работы WAL в PostgreSQL. В лучшем случае, это лишь отправная точка.

Деталей действительно много — в одном только xlog.c почти 9000 строк кода. А ведь помимо него еще есть xloginsert.c, xlogrecovery.c и другие. Рассмотреть их все в рамках одного поста не представляется возможным, да и лишено особого смысла. Ведь со временем информация устареет.

В качестве дополнительных материалов можно рекомендовать:

- Главу 30 официальной документации PostgreSQL;
- Главу 9 книги «The Internals of PostgreSQL»;
- Главу 10 книги «PostgreSQL 15 изнутри»;
- Лекцию 20 из курса CMU Intro to Database Systems;
- Лекцию 10 из курса CMU Advanced Database Systems;
- Файл src/backend/access/transam/README;

Ну и, конечно же, читать код и комментарии к нему.

Дополнение: В продолжение темы см посты Внутренности PostgreSQL: XID wraparound, Внутренности PostgreSQL: карта видимости, Внутренности PostgreSQL: кэш системного каталога и далее по ссылкам.

Метки: PostgreSQL, Алгоритмы, СУБД.