1

1

Обновление версий PostgreSQL

Обновления PostgreSQL

- Минорные обновления PostgreSQL выпускаются раз в 3 месяца или чаще
- Мажорные версии PostgreSQL выпускаются раз в год, в них добавляются новшества
- Начиная с 10 версии основная версия обозначается первым числом до точки
  - В версии 18.4 мажорная версия 18, минорная 4.
- Установка минорных обновлений менее рискованна, чем продолжение работы со старой минорной версией
- В минорные обновления Tantor Postgres добавляются новшества, которые не меняют формат хранения файлов кластера
  - проводятся по процедуре минорных обновлений

Обновления PostgreSQL

Минорные обновления PostgreSQL выпускаются раз в 3 месяца или чаще. Чаще выпускаются, если нужно устранить критичную ошибку. Даты выхода минорных обновлений: https://www.postgresql.org/developer/roadmap/

Мажорные версии PostgreSQL выпускаются раз в год, в них добавляются новшества. Начиная с 10 версии основная версия обозначается первым числом до точки. В версии 18.4 мажорная версия 18, минорная 4.

Для минорных обновлений достаточно установить обновлённые бинарные файлы и перезапустить экземпляр. Сообщество считает, что установка минорных обновлений менее рискованна, чем продолжение работы со старой минорной версией.

Для обновления мажорной версии не нужно устанавливать все промежуточные обновления, достаточно установить целевую версию и выполнить процедуру обновления. При этом рекомендуется прочитать разделы "Migration to Version" раздела Release Notes всех версий между исходной и целевой версией включительно. Обычно, такой раздел имеется у первой мажорной версии (17.0, 18.0, 19.0) и некоторых минорных. Например, после миграции на версию 18.2 или более новую с предыдущих версий, понадобится пересоздать индексы на столбцы типа ltree (https://www.postgresql.org/docs/18/release-18-2.html). При переходе с версий до 17.1 уделить внимание секционированным таблицам, в которых отсоединялись секции (https://www.postgresql.org/docs/17/release-17-1.html).

При обновлении Tantor Postgres можно обратиться в техническую поддержку и получить рекомендации по обновлению.

https://www.postgresql.org/support/versioning/

Минорные обновления PostgreSQL

- Распространяются в виде полноценных дистрибутивов - пакетами `deb` и `rpm`
- Если устанавливается не выдавая ошибок, то деинсталлировать установленный пакет не нужно
- После установки достаточно перезапустить экземпляр
- В названии службы Tantor Postgres, создаваемой по умолчанию, имеется название сборки: **se**, **se1c**, **certified**, **certified-2**, **be**, **free**, **polar**

```
psql -qtAc "select tantor_version()"
 Tantor Certified Edition 18.3.0
dpkg -i tantor-se-server-18_18.3.0.pgo_amd64.deb
pg_ctl stop
sudo systemctl start tantor-se-server-18
sudo systemctl enable tantor-se-server-18
psql -qtAc "select tantor_version()"
 Tantor Special Edition 18.3.0.pgo
```


Минорные обновления PostgreSQL

Для установки минорного обновления нужно:
1) скачать дистрибутив новой минорной версии. Минорные обновления распространяются в виде полноценных дистрибутивов - пакетами `deb` и `rpm` и устанавливаются обычным образом.
2) установить пакет `deb` или `rpm`. Если устанавливается не выдавая ошибок, то деинсталлировать установленный пакет не нужно.
3) перезапустить экземпляр. Можно выполнить контрольную точку, остановить экземпляр и запустить службу.
В названии службы Tantor Postgres, создаваемой по умолчанию, имеется название сборки: **se**, **se1c**, **certified**, **certified-2**, **be**, **free**, **polar**.

Сборки LTO и PGO

- Начиная с версии 18.3 Tantor выпускает сборки "pgo" (Profile Guided Optimization) в дополнение к LTO

```
postgres@tantor:~$ pg_config | grep clang
CONFIGURE =  '--prefix=/opt/tantor/db/18' ... -DTANTOR_SE' ...
'CLANG=/usr/bin/clang-13' ...
postgres@tantor:~$ pg_config | grep pgo
LDFLAGS = -Wl,-z,relro -Wl,-z,now -flto=auto -ffat-lto-objects -fprofile-
use=/builds/database/tantor-db/.pgo-profile -fprofile-correction -Wno-
error=coverage-mismatch -Wno-error=missing-profile -L/usr/local/lib/zstd
postgres@tantor:~$ psql -qtAc "select tantor_version()"
 Tantor Special Edition 18.3.0.pgo
postgres@tantor:~$ pgbench -T 30 -c 2 2> /dev/null | grep tps
tps = 772.190215 (without initial connection time)
postgres@tantor:~$ psql -qtAc "select tantor_version()"
 Tantor Special Edition 18.3.0
postgres@tantor:~$ pgbench -T 30 -c 2 2> /dev/null | grep tps
tps = 734.166745 (without initial connection time)
```


Сборки LTO и PGO

Начиная с версии 18.3 Tantor выпускает сборки "pgo" (Profile Guided Optimization). Сначала идёт компиляция с оптимизациями на основе текста исходного кода - LTO (Link Time Optimization). На втором шаге на созданном и запущенном экземпляре запускается тестовая нагрузка и собираются показатели (статистика) работы - "профиль" нагрузки. На третьем шаге выполняется компиляция с учётом собранных показателей. При выборе метода оптимизации частей кода, компилятор меньше полагается на текст, он использует собранную статистку, что помогает оптимизировать более эффективно части кода. Эффективность компиляции PGO зависит от качества теста, который симулирует рабочую нагрузку.

PGO не заменяет LTO, а дополняет его.

Для PGO нужен атрибут для кода `__attribute__((no_profile))`, который появился в `clang-13`. Tantor Postgres 18 и Tantor Polar 15 компилируются с использованием `clang-13`. Предыдущие версии Tantor Postgres компилировались `clang-11` и использовали LTO.

Проверить, как скомпилирован PostgreSQL, можно командой:

```
pg_config | grep clang
CONFIGURE =  '--prefix=/opt/tantor/db/18' ... 'CLANG=/usr/bin/clang-13'
```

По умолчанию, PostgreSQL компилируется с LTO. PGO не используется из-за сложности: нужна двойная компиляция и тест.

Andres Freund: "Я заметил довольно существенное ускорение при использовании PGO, но тестировал только очень специфические нагрузки. Насколько я помню, прирост составил более 15% в параллельном чтении в `pgbench`."

https://www.postgresql.org/message-id/20230127230752.3tc5crhrdgeoker2%40awork3.anarazel.de

Минорные обновления с деинсталляцией

- При деинсталляции, установленная служба с именем по умолчанию, будет удалена
- У разных сборок директория PGDATA по умолчанию может иметь другое название и нужно будет отредактировать пути в файле службы по умолчанию

```
psql -c "select tantor_version()"
 Tantor Certified Edition 18.3.0
sudo apt list | grep tantor | grep installed
tantor-se-server-18/now 18.3.0 amd64 [installed,local]
sudo apt purge tantor-se-server-18 -y
sudo dpkg -i tantor-certified-server-18_18.3.0_amd64.deb
pg_ctl restart
 Tantor Certified Edition 18.3.0
echo $PGDATA
/var/lib/postgresql/tantor-se-18/data
cat /usr/lib/systemd/system/tantor-certified-server-18.service | grep data
Description=Tantor Certified Edition database server 18
Environment=PGDATA=/var/lib/postgresql/tantor-certified-18/data
```


Минорные обновления с деинсталляцией

При установке другой сборки могут выдаваться ошибки:

```
dpkg: error processing archive tantor-certified-server-18_18.3.0_amd64.deb (--install):
 trying to overwrite '/opt/tantor/db/18/bin/check_unique_constraint.py', which is also in package tantor-se-server-18 18.3.0.pgo
```

В этом случае, нужно деинсталлировать установленный пакет и только потом устанавливать новый. При деинсталляции, установленная служба с именем по умолчанию, будет удалена вместе с файлом службы и символическими ссылками в поддиректориях `/etc/systemd/system/`, указывающими на файл службы.

Если при смене сборки потребуется обновление кластеров баз данных, а не просто перезапуск экземпляров, нужно сохранить (скопировать) директорию с программным обеспечением `/opt/tantor/db/`*`версия`*, так как она понадобится при обновлении утилитой `pg_upgrade`.

У другой сборки, название директории PGDATA по умолчанию, может быть другим. После установки новой сборки нужно будет отредактировать пути в файле службы по умолчанию и включить службу. После этого остановить экземпляр и запустить службу.

Если программное обеспечение обслуживало несколько экземпляров, то после установки новой сборки их службы нужно перезапустить (или остановить и запустить) и убедиться, что службы перезапустились.

Мажорные обновления

- Способы:
  - выгрузка и загрузка на логическом уровне утилитами `pg_dumpall`, `pg_dump`
  - использование утилиты `pg_upgrade`
  - использование логической репликации
- Если новая версия софта не может работать с кластером, то попытка запуска экземпляра выдаст ошибку
  - Попытка запуска кластера софтом новой версии не портит файлы кластера

```
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17/data
waiting for server to start....
FATAL:  database files are incompatible with server
DETAIL:  The data directory was initialized by PostgreSQL version 17, which is
not compatible with this version 18.3.
pg_ctl: control file appears to be corrupt
```


Мажорные обновления

Если новая версия софта не может работать с кластером, то попытка запуска экземпляра выдаст ошибку:

```
pg_ctl: control file appears to be corrupt
FATAL:  database files are incompatible with server
DETAIL:  The data directory was initialized by PostgreSQL version 17, which
is not compatible with this version 18.3.
```

Попытка запуска кластера софтом новой версии не портит файлы кластера.

Способы установить мажорные обновления:

1) выгрузка и загрузка на логическом уровне. Кластер, в который будут загружаться данные, нужно создать утилитой `initdb` новой версии софта.

Выгрузить данные можно утилитами `pg_dumpall` и/или `pg_dump` от новой версии. Эти утилиты способны выгружать данные из любой предыдущей версии PostgreSQL, вплоть до 9.2. Также можно выгружать данные утилитами `pg_dumpall` и/или `pg_dump` от прежней версии.

Этот способ наиболее простой, но время обновления зависит от размера кластера баз данных.

Перед выгрузкой данных нужно остановить обслуживание клиентов, так как эти изменения будут потеряны. С этого момента начинается простой приложений.

2) использование утилиты `pg_upgrade`. Этот способ более быстрый, чем выгрузка и загрузка на логическом уровне. При использовании опций `--link` (`-k`) или `--clone` в пределах одной точки монтирования, время обновления не зависит от размера кластера баз данных.

3) использование логической репликации. Этот способ требует больше всего ручных действий, но он даёт минимальный простой обслуживания приложений.

https://docs.tantorlabs.ru/tdb/ru/18_3/se/upgrading.html

Подготовка кластера баз данных

- Кластер баз данных новой версии нужно заранее создать
- Параметры нового кластера не зависят от параметров исходного кластера
- Если данные перегружаются между кластерами на одном хосте, то два кластера должны работать на разных портах
- Проверить, какие библиотеки и расширения установлены на старом кластере и установить эти библиотеки и расширения на новом кластере
- Скопировать параметры конфигурации со старого кластера из файлов: `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`

```
initdb -k -g
echo "port=5433" >> $PGDATA/postgresql.auto.conf
pg_ctl start
```


Подготовка кластера баз данных

Для миграции нужен кластер баз данных новой версии, который нужно заранее создать.

Так как данные перегружаются на логическом уровне, то параметры его создания могут быть любыми и не зависят от параметров исходного кластера. Пример команды создания нового кластера:

```
initdb -k -g
echo "port=5433" >> $PGDATA/postgresql.auto.conf
pg_ctl start
```

Для логической репликации из физической реплики может создаваться клон утилитой `pg_createsubscriber`. Клон затем обновляется утилитой `pg_upgrade`, для которой, в свою очередь, нужно создать новый кластер утилитой **`initdb`**.

Если данные перегружаются между кластерами на одном хосте, то два кластера должны работать на разных портах.

Стоит скопировать параметры конфигурации со старого кластера из файлов: `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`. Проверить по Release Notes нет ли параметров, которые были убраны в новой версии.

Проверить, какие библиотеки и расширения установлены на старом кластере и установить эти библиотеки и расширения на новом кластере. Для этого можно использовать команды **`\dx`** и **`\dconfig *libraries`**

Расширения, установленные в базах данных исходного кластера будут выгружены командами `CREATE EXTENSION`, но скрипты, управляющие файлы расширений, библиотеки должны быть установлены на новом кластере до перегрузки. Расширения могут выдавать ошибку при установке, если их библиотеки не были загружены при запуске экземпляра:

```
postgres=# create extension pg_stat_kcache;
ERROR:  This module can only be loaded via shared_preload_libraries
```

Часть расширений может быть убрана в новой версии PostgreSQL. Например, в PostgreSQL 17.0 было убрано расширение adminpack. Об этом было написано в Release Notes: https://docs.tantorlabs.ru/tdb/en/17_10/se/release-17.html : "Remove adminpack contrib extension. This was used by now end-of-life pgAdmin III".

Выгрузка и загрузка на логическом уровне

- Этот способ наиболее простой, но время обновления зависит от размера кластера баз данных
- Выгружать лучше утилитой `pg_dumpall` от новой версии
- `pg_dumpall` выгружает общие объекты кластера: команды создания ролей, табличных пространств, баз данных
- Данные каждой базы данных выгружаются согласованно на момент подсоединения к этой базе
- Перед выгрузкой данных нужно остановить обслуживание клиентов, так как эти изменения будут потеряны
- Параметр `psql -X` указывает не читать файл `~/.psqlrc`

```
pg_dumpall --statistics --quote-all-identifiers -p 5433 | psql -X
```


Выгрузка и загрузка на логическом уровне

Перед выгрузкой данных нужно остановить обслуживание клиентов, так как эти изменения будут потеряны. `pg_dumpall` выгружает определения общих объектов кластера: пользователей, права пользователей на изменение параметров конфигурации, определения табличных пространств, базы данных. Определения объектов выгружаются в виде команд их создания и изменения.

Команды создания баз данных формируются непосредственно перед командой `\connect` и выгрузкой содержимого баз данных. База данных `template0` и команда её создания не выгружается. Первой выгружается баз данных `template1`. Пример команд:

```
\restrict QlTrAbxuJBTrcfFSsoFIa5Lrt
-- Dumped from database version 17.5
-- Dumped by pg_dump version 18.3
SET statement_timeout = 0;
-- Name: db01; Type: DATABASE; Schema: -; Owner: postgres
CREATE DATABASE db01 WITH TEMPLATE = template0 ENCODING = 'UTF8'
LOCALE_PROVIDER = libc LOCALE = 'en_US.UTF-8';
ALTER DATABASE db01 OWNER TO postgres;
\connect db01
SET statement_timeout = 0;
-- PostgreSQL database dump complete
\unrestrict QlTrAbxuJBTrcfFSsoFIa5Lrt
```

Выгружать лучше утилитой `pg_dumpall` от новой версии. `pg_dumpall` выгружает в единственном формате - текстового командного файла (скрипта). Для формирования скрипта вызывается функционал `pg_dump` для каждой базы данных в кластере, кроме `template0` . Данные каждой базы данных выгружаются согласованно на момент подсоединения к этой базе. Разные базы данных выгружаются на разные моменты времени. `pg_dumpall` подсоединяется к каждой базе по отдельности. Если настроена аутентификация по паролю, то каждый раз будет запрашиваться пароль. Стоит настроить аутентификацию в файлах `pg_hba.conf` (`trust` или `peer`) кластеров или переменную окружения `PGPASSWORD` или файл `~/.pgpass` так, чтобы команда `\connect база` подсоединяла к базе данных без пароля. Параметры командной строки `-h` `-p` и другие наследуются командой и перекрывают переменные окружения `PGHOSTADDR`, `PGPORT`.

https://docs.tantorlabs.ru/tdb/ru/18_3/se/libpq-envars.html

Параллельная выгрузка и загрузка

- Для ускорения можно перегрузить общие объекты кластера опцией `--globals-only`
- выгружать базы данные параллельно с опциями `-j N -F`
- выгружать статистику опцией `--statistics`
- загружать дамп в формате директории параллельно с опциями `-j N -F d`
- `--transaction-size=1000` позволяет объединять в транзакцию заданное параметром число команд

```
postgres@tantor:~$ pg_dumpall -p 5433 --globals-only | psql -X
rm -rf dir
pg_dump -p 5433 -C -j 4 -F d --statistics --quote-all-identifiers -f dir -d db01
pg_restore -c --if-exists --transaction-size=1000 -C -j 2 -F d -d template1 db01
rm -rf dir
```


Параллельная выгрузка и загрузка

Выгрузка баз данных выполняется последовательно одним процессом без распараллеливания. Для ускорения выгрузки можно перегрузить пользователей, права пользователей на изменение параметров конфигурации, определения табличных пространств командой:

```
pg_dumpall -p 5433 --globals-only | psql -X
```

После этого останавливать обслуживание приложений, подсоединяющихся к базам данных и выгружать базы данные командой:

```
pg_dump -C -j 4 -F d --statistics --quote-all-identifiers -f dir -d db01
```

Опция `-C` выгружает команду создания базы данных.

Опция `--statistics` выгружает статистику и экономит время на сбор статистики.

Опция `--quote-all-identifiers` рекомендуется для предотвращения проблем, возникающих при изменении набора зарезервированных слов в разных версиях PostgreSQL.

Опция `-j число` указывает использовать при выгрузке заданное `число` рабочих процессов, которые будут параллельно выгружать содержимое базы данных. Одну таблицу выгружает один рабочий процесс, несколько процессов не могут выгружать параллельно данные одной таблицы или секции таблицы. Эта опция может использоваться только с опциями `-F d -f имя`. Эти опции указывают, что формат дампа - директория и `имя` директории. В отличии от текстового формата, приходится выгружать данные базы в файлы, что занимает место на диске. Также нельзя использовать пайп и начинать загрузку одновременно с выгрузкой. Из-за этого указывать число рабочих процессов меньше, чем 3 не имеет преимуществ по сравнению с использованием пайпа с текстовым форматом.

Опции `-c --if-exists` формируют команды удаления `DROP .. IF EXISTS` выгружаемых объектов перед командами создания объектов.

По умолчанию, используется автофиксация - после каждой команды выполняется COMMIT.

Опция `--transaction-size=1000` позволяет объединять в транзакцию заданное параметром число команд. При большом числе объектов ускоряет загрузку. В экземпляре должно быть достаточно свободных блокировок.

Выгрузка и загрузка статистики

- появилась в 18 версии и позволяет загружать статистику
- статистика по таблицам, индексам и столбцам

```
drop table if exists tab;
create table tab (col text);
insert into tab values ('CCC'),
('BBB'), ('AAA'), ('D'), ('D');
analyze tab;
```

```
SELECT * FROM
 pg_restore_relation_stats(
'version', 180003,
'schemaname', 'public',
'relname', 'tab',
'relpages', 1,
'reltuples', '5'::real,
'relallvisible', 0,
'relallfrozen', 0);
```

```
pg_dump -t "tab" --statistics-only
```

```
SELECT * FROM
 pg_restore_attribute_stats(
'version', 180003,
'schemaname', 'public',
'relname', 'tab',
'attname', 'col',
'inherited', false,
'null_frac', '0'::real,
'avg_width', 3,
'n_distinct', '-0.8'::real,
'most_common_vals', '{D}',
'most_common_freqs', '{0.4}'::real[],
'histogram_bounds', '{AAA,BBB,CCC}',
'correlation', '0.6'::real);
```


Выгрузка и загрузка статистики

Выгрузка и загрузка статистики уменьшает время на сбор статистики после загрузки. Для нескольких сотен тысяч таблиц сбор статистики может занять 10 минут, при настройках по умолчанию (`default_statistics_target = 100`). Создание жёстких ссылок занимает в 2 раза меньше времени. Опция выгрузки и загрузки статистики появилась **в 18 версии** и позволяет выгружать статистику с предыдущих версий PostgreSQL, начиная с версии 9.2. Опция формирует команды вызова функций `pg_restore_relation_stats()`, `pg_restore_attribute_stats()`, а **в 19 версии** к ним добавилась функция `pg_restore_extended_stats()`.

В 18 версии появились параметры у утилит `pg_upgrade --no-statistics`, `pg_dump` и `pg_dumpall --no-statistics`, `--statistics`, `--statistics-only`.

Выгружается статистика по отношениям (таблицам, индексам): число строк (relallvisible), число блоков данных в которых нет старых версий строк (relallvisible), число блоков данных, в которых все строки ещё и заморожены (relallfrozen). Число блоков данных тоже выгружается, но планировщик полагается на размеры файлов данных.

Выгружается статистика по столбцам: доля пустых значений в столбце (null_frac), доля уникальных и непустых значений (если n_distinct отрицательно), наиболее часто встречающиеся значения (`most_common_vals`) , гистограмму распределения значений. Максимальное число значений и корзин в гистограмме, по умолчанию, определяется параметром `default_statistics_target`.

Использование pg_upgrade

- перегружает данные из таблиц системного каталога на логическом уровне

```
Creating dump of global objects
"/opt/tantor/db/18/bin/pg_dumpall" --host /var/lib/postgresql/tantor-se-18 --port
50432 --username postgres --globals-only --quote-all-identifiers --binary-upgrade
--verbose --no-sync -f "/var/lib/postgresql/tantor-se-
18/data1/pg_upgrade_output.d/20260627T092815.022/dump/pg_upgrade_dump_globals.sql"
>> "/var/lib/postgresql/tantor-se-
18/data1/pg_upgrade_output.d/20260627T092815.022/log/pg_upgrade_utility.log" 2>&
Creating dump of database schemas
"/opt/tantor/db/18/bin/pg_dump" --host /var/lib/postgresql/tantor-se-18 --port
50432 --username postgres --no-data --sequence-data --verbose --quote-all-
identifiers --binary-upgrade --format=custom --statistics --no-sync --
file="/var/lib/postgresql/tantor-se-
18/data1/pg_upgrade_output.d/20260627T092815.022/dump/pg_upgrade_dump_1.custom"
'dbname=template1' >> "/var/lib/postgresql/tantor-se-
18/data1/pg_upgrade_output.d/20260627T092815.022/log/pg_upgrade_dump_1.log" 2>&1
```


Использование pg_upgrade

Выгрузка и загрузка данных на логическом уровне зависит от объема данных. Длительность перегрузки данных может быть неприемлемо большой. В таком случае, удобна утилита `pg_upgrade`.

Параметры локализации нового кластера (создаётся заранее утилитой `initdb`) не важны, они будут заменены параметрами исходного кластера. Параметры **подсчета контрольных сумм и размер WAL-файлов должны быть такими же, как у исходного кластера**, утилита `pg_upgrade` это проверит.

Новшества в мажорных версиях могут менять структуру таблиц системных каталогов баз данных и общих таблиц кластера и, обычно, не меняют структуру блоков. Поэтому, обновление можно выполнить быстро, используя файлы данных исходного кластера.

При переходе с ванильного PostgreSQL, Tantor BE, Tantor Free на сборки с 64-битным счётчиком транзакций (Tantor SE, SE 1C, Certified 1), утилита `pg_upgrade` не обновляет блоки. Добавление зоны special размером 16 байт в конец блоков данных выполняются после миграции, в процессе работы серверных процессов с блоками таблиц. Благодаря этому, длительность миграции не зависит от размера кластера.

Утилита `pg_upgrade` перегружает данные из **таблиц системного каталога** каждой базы и **общих таблиц** кластера на логическом уровне, вызывая утилиты с параметрами:

`pg_dumpall --globals-only --quote-all-identifiers --binary-upgrade --no-sync`

и

`pg_dump --no-data --sequence-data --verbose --quote-all-identifiers --binary-upgrade --format=custom --statistics --no-sync`

Выгрузка выполняется в формате **`custom`**, а не `directory`, так как выгружаются **только метаданные**. Для каждой базы данных создаётся файл **`pg_upgrade_dump_OID.custom`** в директории **`PGDATA_NEW/pg_upgrade_output.d/дата_и_время/dump`**.

`pg_upgrade` создаёт директорию **`pg_upgrade_output.d`** в `PGDATA` нового кластера. Эта директория будет удалена, если `pg_upgrade` успешно доработает до конца и не использовался параметр `--retain`. Если не доработает, то файлы в этой директории могут пригодиться для выяснения причины ошибки. Параметр позволяет изучить, какие команды и действия выполняет утилита `pg_upgrade` в процессе обновления.

Использование pg_upgrade

- файлы данных копируются, перемещаются, клонируются или линкуются на уровне файловой системы:
  - `--copy` или `--copy-file-range`
  - `--link`
  - `--clone` на `ext4` не работает
  - `--swap` перемещение директорий
- `--check` запускает утилиту в тестовом режиме
- `--retain` не удаляет директорию с логами `pg_upgrade_output.d`

```
postgres@tantor:~$ rm -rf /var/lib/postgresql/tantor-se-18/data
/opt/tantor/db/18/bin/initdb -k -g -D /var/lib/postgresql/tantor-se-18/data
/opt/tantor/db/18/bin/pg_upgrade --check --link --old-bindir=/opt/tantor/db/17/bin
--new-bindir=/opt/tantor/db/18/bin --old-datadir=/var/lib/postgresql/tantor-se-
17/data --new-datadir=/var/lib/postgresql/tantor-se-18/data
...
*Clusters are compatible*
```


Использование pg_upgrade

Длительность перегрузки не зависит от размера кластера, но зависит от числа объектов в базах кластера. Для сотни тысяч небольших (не много столбцов и файлов) таблиц, ориентировочно, десяток минут. Формат блоков данных, обычно, не меняется и данные таблиц можно не выгружать, а использовать файлы исходного кластера.

Файлы данных утилитой:

1) копируются, параметр `--copy` или `--copy-file-range` или

2) дублируются путем создания жёстких ссылок (hardlink), параметр `--link` или

3) клонируются, параметр `--clone`, на файловых системах, где есть функционал reflink (xfs и btrfs) или

4) перемещаются в директориях, параметр `--swap`.

Для `--link` и `--clone` директории старого и нового кластера должны находиться на одной и той же файловой системе.

Опция `--check` позволяет проверить, сможет ли потенциально утилита обновить кластер с теми параметрами, которые будут переданы утилите. Обновляемый кластер не нужно останавливать , то есть проверка может выполняться без простоя. Кластер новой версии должен находиться в остановленном состоянии. Если `--check` выдаёт:

`*Clusters are compatible*`

это не гарантирует, что при реальном запуске не будет ошибок и утилита не прервётся на каком-то этапе.

Обновление мастера утилитой pg_upgrade

- запускает экземпляры на порту 50432
- создаёт директорию pg_upgrade_output.d в PGDATA нового кластера
- --no-statistics - статистика не будет выгружаться
- --sync-method=syncfs синхронизировать файловые системы нового кластера

```
postgres@tantor:~$ /opt/tantor/db/18/bin/pg_upgrade --link -b
/opt/tantor/db/17/bin -B /opt/tantor/db/18/bin -d /var/lib/postgresql/tantor-se-
17/data -D /var/lib/postgresql/tantor-se-18/data -j 4
Upgrade Complete
----------------
Some statistics are not transferred by pg_upgrade.
Once you start the new server, consider running these two commands:
    /opt/tantor/db/18/bin/vacuumdb --all --analyze-in-stages --missing-stats-only
    /opt/tantor/db/18/bin/vacuumdb --all --analyze-only
Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh
```


Обновление мастера утилитой pg_upgrade

pg_upgrade подключается к новому и старому кластеру по несколько раз и стоит настроить аутентификацию без ввода пароля.

По умолчанию, pg_upgrade запускает экземпляры на порту 50432, а не на портах, которые указаны в их параметрах конфигурации, чтобы избежать подключения клиентов. Экземпляры старого и нового кластера будут запускаться и останавливаться (в режиме smart) по очереди на порту 50432. Файл Unix-сокета будет создаваться и удаляться в директории из которой запускается утилита pg_upgrade. Для запуска и остановки используется утилита:

```
pg_ctl -w -D "/var/lib/postgresql/tantor-be-18/data" -o "-p 50432 -b  -c
listen_addresses='' -c unix_socket_permissions=0700 -c
unix_socket_directories='директория запуска утилиты'" start >>
"/var/lib/postgresql/tantor-se-
18/data1/pg_upgrade_output.d/20260627T092815.022/log/pg_upgrade_server.log"
2>&1
```

По умолчанию, синхронизирует каждый файл, что может занять время, при большом числе файлов. Можно использовать параметр --sync-method=syncfs, чтобы синхронизировать файловые системы нового кластера, в которых расположена новая директория PGDATA, и на которые указывают символическая ссылка PGDATA/pg_wal и символические ссылки в PGDATA/pg_tblspc.

Параметр --jobs (или -j) в значении больше 1, позволит обрабатывать параллельно несколько баз данных и табличных пространств. Можно установить в число ядер процессоров. Метаданные каждой базы выгружается одним процессом pg_dump в формате custom, так как распараллеливание выгрузки метаданных не имеет преимуществ.

Выгрузка и загрузка статистики появилась в 18 версии. Статистика может выгружаться из предыдущих версий PostgreSQL, начиная с версии 9.2, но загружаться может только в PostgreSQL **18 версии** и более новых. При выгрузке статистики стоит принять во внимание, что статистика наиболее часто встречающихся значений в столбцах (Most Common Values) появились в **13 версии**.

Если возникают ошибки с выгрузкой или загрузкой статистики, можно использовать параметр --no-statistics, тогда статистика не будет выгружаться. Длительность сбора статистики зависит от числа таблиц в базах данных кластера.

https://habr.com/ru/companies/tantor/articles/1044730/

Обновление реплик и pg_upgrade

- При обновлении мастера в режиме `--link` можно утилитой `rsync` обновить реплики
  - При использовании других режимов реплики придётся создать заново
  - На репликах новые директории PGDATA должны быть пустыми или отсутствовать
  - Пути к директориям PGDATA, табличных пространств и WAL на хосте мастера и реплик должны быть одинаковыми
  - После `pg_upgrade --link` нельзя запускать экземпляр мастера до обновления одной из реплик утилитой `rsync`

```
postgres@tantor:~$ sudo apt install rsync
cd /var/lib/postgresql
rsync --archive --delete --hard-links --size-only --no-inc-recursive --verbose --
info=progress2 /var/lib/postgresql/tantor-se-17/data /var/lib/postgresql/tantor-
se-18/data postgres@standbyhost:/var/lib/postgresql
```


Обновление реплик и pg_upgrade

При работе физической репликации, мастер и физические реплики должны иметь одинаковую мажорную версию. При обновлении мастера в режиме **--link** можно относительно быстро обновить реплики. При использовании других режимов реплики придётся пересоздать. После обновления мастера утилитой `pg_upgrade --link` нельзя запускать экземпляр мастера. На репликах новые директории `PGDATA` должны быть пустыми или отсутствовать. Пути к директориям `PGDATA`, табличных пространств и WAL на хосте мастера и реплик должны быть одинаковыми. Нужно:

- остановить экземпляры реплик, если они ещё не были остановлены.
- заранее скопировать или переместить файлы конфигурации реплик и, если нужно, отредактировать параметры конфигурации, убрав устаревшие параметры
- на хосте мастера перейти в директорию, в которой находятся директории старой и новой `PGDATA`:

```
cd /var/lib/postgresql
rsync --archive --delete --hard-links --size-only --no-inc-recursive --
verbose --info=progress2 /var/lib/postgresql/tantor-se-17/data
/var/lib/postgresql/tantor-se-18/data postgres@standbyhost:/var/lib/postgresql
```

Параметром `rsync --dry-run` можно проверить, что будет делать `rsync`.

Команда `rsync` выполняется быстро, создавая жёсткие ссылки на хосте реплики. Затем копируются файлы, которые отсутствуют на реплике или отличаются по размеру. Число таких файлов невелико, это файлы нежурналируемых таблиц.

Если есть табличные пространства, то выполнить ту же команду для директорий каждого табличного пространства. Если `PGDATA/pg_wal` является символической ссылкой, то запустить `rsync` и на директории с файлами журналов.

Ошибки при обновлении pg_upgrade

- Если использовались **--link** или --swap, то, после попытки запуска нового кластера, старый кластер не запустится
- Если в режиме **--link** hardlinks были созданы и новый кластер не запускался, то можно убрать расширение .old у управляющего файла исходного кластера PGDATA/global/pg_control.old чтобы его запустить

```
...
Checking for external indexes                               ok
Creating dump of global objects                             ok
Creating dump of database schemas
*failure*

Consult the last few lines of "/var/lib/postgresql/tantor-se-18-
replica/data/pg_upgrade_output.d/20260517T204747.409/log/pg_upgrade_dump_1.log"
for the probable cause of the failure.
Failure, exiting
```


Ошибки при обновлении pg_upgrade

Даже, если **при проверке** с параметром --check было выдано сообщение:
Clusters are compatible
гарантий того, что утилита успешно доработает до конца нет. Поэтому, важно иметь возможность вернуться к старому кластеру, чтобы можно было его запустить.

После выполнения команд rsync можно запускать экземпляр мастера.

Если реплика не одна, то можно, не запуская экземпляр обновленной реплики (запущен только экземпляр мастера), запустить на хосте реплики rsync для обновления директорий других реплик. После этого восстановить скопированные заранее файлы конфигурации реплик и можно запускать их экземпляры.

Если на репликах были слоты репликации (каскадная репликация), то надо будет создать на репликах физические слоты. Логические слоты репликации, которые стало возможным создавать на репликах **в 17 версии**, будут сохранены.

Если не использовались параметры **--link** или --swap, то в директории прежнего кластера ничего не меняется и его экземпляр можно запустить.

Если использовался параметр --swap, то запустить прежний кластер можно только, если утилита прекратила работу до выдачи сообщения, что запускать старый кластер больше небезопасно.

Опции **--link** и --swap **опасны тем, что после попытки запуска нового кластера, старый кластер не запустится**.

Если использовался параметр **--link** и утилита прекратила работу до создания hardlinks, то экземпляр исходного кластера можно запустить. Если hardlinks были созданы, но попыток запустить экземпляр на новом софте не было, то **можно убрать расширение .old у управляющего файла в директории исходного кластера OLD_PGDATA/global/pg_control.old и запустить экземпляр старого кластера**.

Обновление через логическую репликацию

- Логическая репликация появилась в 10 версии
- Нужно место под новый кластер, а при использовании `pg_upgrade` с параметром `--link` место не нужно
- Все таблицы во всех базах данных исходного кластера должны иметь первичные ключи, либо нужно будет поменять свойство `REPLICA IDENTITY` таблиц, у которых нет первичных ключей
- При переходе с 17 версии можно использовать утилиту `pg_createsubscriber`
- Реплицируются только изменения в таблицах
- Сложность процедуры: много шагов

Обновление через логическую репликацию

Если `pg_upgrade` не может обновить кластер или длительность обновления должна быть меньше, чем выполняется `pg_upgrade`, можно использовать обновление через логическую репликацию. Логическая репликация появилась в **10 версии** PostgreSQL, это минимальная версия, с которой можно обновиться с помощью встроенной логической репликацией.

Недостатки обновления через логическую репликацию:

1) нужно место под новый кластер, а при использовании `pg_upgrade` с параметром `--link` место не нужно.

2) все таблицы во всех базах данных исходного кластера должны иметь первичные ключи, либо нужно будет поменять свойство `REPLICA IDENTITY` таблиц, у которых нет первичных ключей.

3) начальная загрузка данных в таблицы нового кластера выполняется долго. Выгрузка данных удерживает горизонт очистки и автовакуум не может удалять старые версии строк в исходном кластере.

Если исходный кластер **17 и более новой версии**, то на исходном кластере можно использовать физическую реплику, преобразовать её в клон утилитой `pg_createsubscriber`. Утилита "бесшовно" настроит логическую репликацию всех таблиц. Преобразование выполняется быстро, не зависит от размера таблиц, не приводит к простою.

4) реплицируются только изменения в таблицах. Пока работает логическая репликация, нельзя менять определения объектов в исходном кластере, так как реплицируются только изменения в строках таблиц и команда `TRUNCATE`.

5) **до 19 версии** значения последовательностей не реплицируются и, после остановки обслуживания приложений исходным кластером, значения нужно выгрузить и загрузить в базы нового кластера.

Материализованные представления, большие объекты (`lo`) не реплицируются. После конвертации физической реплики в клон в них не стоит вносить изменения.

Команда `TRUNCATE tab CASCADE` усечёт зависимые таблицы на подписчике только, если они включены в публикацию, что, обычно, и делается.

6) большая сложность обновления. Если исходный кластер имеет подписчиков, это нужно будет учитывать и перенести подписчиков на обновлённый клон.

Последовательности

- Последовательности являются отношениями
- У отношений общее пространство имён

```
CREATE SEQUENCE t_id_seq;
CREATE TABLE t (id BIGINT NOT NULL DEFAULT nextval('t_id_seq'));
ALTER SEQUENCE t_id_seq OWNED BY t.id;
```

- что эквивалентно:

```
CREATE TABLE t (id BIGSERIAL);
\d t                    Table "public.t"
 Column |  Type  | Nullable |        Default
--------+--------+----------+------------------------------
 id     | bigint | not null | nextval('t_id_seq'::regclass)
```

- Пример использования последовательности:

```
select nextval('t_id_seq'), currval('t_id_seq');
 nextval | currval
---------+---------
       1 |       1
```


Последовательности

Последовательности являются "отношениями", объектами постоянного хранения. У отношений общее пространство имён - создать таблицу и последовательность с одинаковым именем в одной схеме нельзя. Последовательности можно создать командой:

```
CREATE temp SEQUENCE t_id_seq;
```

и использовать функциями:

```
select nextval('t_id_seq'), currval('t_id_seq');
```

в определениях столбцов таблиц:

```
CREATE temp TABLE t (id BIGINT NOT NULL DEFAULT nextval('t_id_seq'));
```

Если нужно, чтобы при удалении столбца последовательность удалялась, то можно её привязать к столбцу таблицы:

```
ALTER SEQUENCE t_id_seq OWNED BY t.id;
```

К столбцу можно привязать несколько последовательностей, но использоваться будет та, которая указана в определении столбца таблицы и для `*SERIAL` и для `IDENTITY` столбцов.

Это эквивалентно использованию синтаксиса `SERIAL` и `BIGSERIAL`:

```
CREATE temp TABLE t (id BIGSERIAL);
\d t                        Table "pg_temp_67.t"
 Column |  Type  | Collation | Nullable |            Default
--------+--------+-----------+--------- +------------------------------
 id     | bigint |           | not null | nextval('t_id_seq'::regclass)
```

`SERIAL` и `BIGSERIAL` используют типы `integer` и `bigint`, создают последовательность с уникальным именем, ограничение целостности `NOT NULL`, привязывают последовательность к столбцу.

В 12 версии появились хранимые генерируемые столбцы:

```
GENERATED ALWAYS AS ( выражение ) STORED
```

В 18 версии появились виртуальные (вычисляемые) столбцы:

```
GENERATED ALWAYS AS ( выражение ) VIRTUAL
```

Значение для таких столбцов нельзя задать, хотя ключевое слово `DEFAULT` можно указать.

До 18 версии логическая репликация не публиковала генерируемые столбцы. **В 18 версии** , по умолчанию, генерируемые столбцы не публикуются и при миграции их не имеет смысла публиковать. Хранимые генерируемые столбцы могут публиковаться, если на таблице подписчика используется обычный столбец, а не генерируемый.

Последовательности для IDENTITY столбцов

- В 10 версии появился синтаксис `IDENTITY`

```
CREATE temp TABLE t2 (id BIGINT GENERATED BY DEFAULT AS IDENTITY);
CREATE temp TABLE t3 (id integer GENERATED ALWAYS AS IDENTITY);
select pg_get_serial_sequence('t2', 'id');
```

```
CREATE temp TABLE t1 (id bigint generated always as identity primary key);
INSERT INTO t1 VALUES (1);
ERROR:  cannot insert a non-DEFAULT value into column "id"
DETAIL:  Column "id" is an identity column defined as GENERATED ALWAYS.
HINT:  Use OVERRIDING SYSTEM VALUE to override.
INSERT INTO t1 OVERRIDING SYSTEM VALUE VALUES (1);
INSERT INTO t1 VALUES (DEFAULT);
ERROR:  duplicate key value violates unique constraint "t1_pkey"
DETAIL:  Key (id)=(1) already exists.
INSERT INTO t1 VALUES (DEFAULT);
select * from t1;
  1
  2
```


Последовательности для IDENTITY столбцов

В 10 версии появился синтаксис `IDENTITY` из стандарта ISO SQL:2003.

```
CREATE temp TABLE t (id BIGINT GENERATED BY DEFAULT AS IDENTITY);
```

который эквивалентен `BIGSERIAL` с небольшими различиями. Также есть синтаксис:

```
CREATE temp TABLE t (id integer GENERATED ALWAYS AS IDENTITY);
```

С новым синтаксисом также столбцу добавляется `NOT NULL`, создаётся последовательность, которая привязывается `OWNED BY` к столбцу, благодаря чему последовательность удалится при удалении столбца. Найти имя используемой последовательности можно функцией:

```
select pg_get_serial_sequence('t', 'id');
```

С `BY DEFAULT` можно установить значение явно и использовать ключевое слово `DEFAULT`.

С `ALWAYS` попытка вставить в столбец значение, отличное от `DEFAULT` выдаст ошибку. Однако, для команды **`INSERT`** есть опция, которая позволяет вставить произвольное значение:

```
INSERT INTO t(id) OVERRIDING SYSTEM VALUE VALUES (1);
```

Для команды `UPDATE` такой опции нет.

Различия:

1) на последовательности, используемые с `serial`и `bigserial` нужно предоставлять привилегии `GRANT USAGE ON SEQUENCE имя`, а на последовательности, используемые `IDENTITY` столбцами не нужно:

```
CREATE TABLE t1 (id bigserial primary key);
CREATE TABLE t2 (id bigint generated by default as identity primary key);
CREATE USER alice; GRANT ALL ON TABLE t1, t2 TO alice; SET ROLE alice;
INSERT INTO t2 VALUES (default); INSERT INTO t1 VALUES (default);
RESET ROLE;
ERROR:  permission denied for sequence t1_id_seq
```

2) При создании наследника, наследник будет использовать ту же самую последовательность:

```
CREATE TEMP TABLE t11 (LIKE t1 INCLUDING ALL);
\d t1 \\ \d t11 \\
```

3) с `IDENTITY` столбцом можно не указывать имя последовательности в команде:

```
ALTER TABLE t2 ALTER COLUMN id RESTART WITH 10;
```

но это не существенно.

Последовательности и логическая репликация

- До 19 версии значения последовательностей не реплицируются

```
create table t1 (col bigint generated always as identity primary key);
create table t2 (col bigserial unique not null);
create sequence aaa start with 5 increment by 5;
insert into t1 values (default);
alter sequence aaa restart with 8;
select schemaname || '.' || sequencename seq_name, start_value, last_value,
coalesce(last_value,start_value) set_value from pg_sequences;
     seq_name      | start_value | last_value | set_value
-------------------+-------------+------------+-----------
 public.t1_col_seq |           1 |          1 |         1
 public.t2_col_seq |           1 |            |         1
 public.aaa        |           5 |            |         5
select sequence_name from information_schema.sequences;
 sequence_name
---------------
 t2_col_seq
 aaa
```


Последовательности и логическая репликация

До 19 версии значения последовательностей не реплицируются и, после остановки обслуживания приложений исходным кластером, значения нужно выгрузить и загрузить в базы нового кластера. **В 19 версии** последовательности можно обновить командой `ALTER SUBSCRIPTION имя REFRESH SEQUENCES`.

Пример создания трёх последовательностей разными способами:

```
drop table if exists t1, t2; drop sequence aaa;
create table t1 (col bigint GENERATED ALWAYS AS IDENTITY primary key);
create table t2 (col bigserial unique not null);
create sequence aaa start with 5 increment by 2;
insert into t1 values (default);
```

После переустановки значения, это значение не показывается в представлении `pg_sequences`:

```
alter sequence aaa restart with 8;
select schemaname || '.' || sequencename seq_name, start_value, last_value,
coalesce(last_value,start_value) set_value from pg_sequences;
     seq_name      | start_value | last_value | set_value
-------------------+-------------+------------+-----------
 public.t1_col_seq |           1 |          1 |         1
 public.t2_col_seq |           1 |            |         1
 public.aaa        |           5 |            |         5
```

Нельзя использовать `information_schema.sequences`, так как представление не выдаёт последовательности, обслуживающие `GENERATED {ALWAYS | BY DEFAULT} AS IDENTITY`, которые появились **в 12 версии**:

```
select sequence_name from information_schema.sequences;
 sequence_name
---------------
 t2_col_seq
 aaa
(2 rows)
```

https://docs.tantorlabs.ru/tdb/ru/18_3/se/logical-replication-gencols.html

Текущее значение последовательности

- Текущее значение покажет запрос по имени или `nextval()` или `pg_get_sequence_data()`

```
select * from aaa;
 last_value | log_cnt | is_called
------------+---------+-----------
          8 |       0 | f
select * from pg_get_sequence_data('aaa');
 last_value | is_called
------------+-----------
          8 | f
select currval('aaa');
ERROR:  currval of sequence "aaa" is not yet defined in this session
select nextval('aaa');
 nextval
---------
       8
select nextval('aaa');
 nextval
---------
      13
```


Текущее значение последовательности

Текущее значение покажут:

1) запрос по имени последовательности:

```
select * from aaa;
 last_value | log_cnt | is_called
------------+---------+-----------
          8 |       0 | f
```

2) функция **`select pg_sequence_last_value('aaa')`**. Эта функция неудобна тем, что возвращает пустое значение, если `is_called=false`.

3) функция `pg_get_sequence_data()`, которая появилась только в **18 версии**:

```
select * from pg_get_sequence_data('aaa');
 last_value | is_called
------------+-----------
          8 | f
```

4) функция `nextval()` вернёт `last_value`, если `is_called=false` и установит `is_called=true`. Если `is_called=true`, то функция добавит к `last_value` значение `increment by` (продвинет последовательность до следующего значения) и вернёт это значение.

Так как функция `nextval()` вносит изменение в последовательность, то она **не работает на физических репликах**:

```
ERROR:  cannot execute nextval() in a read-only transaction
```

Функция для считывания текущего значения **`currval()`** не может использоваться, если предварительно в этой сессии не выбрать из последовательности значения функцией **`nextval()`**:

```
select currval('aaa');
ERROR:  currval of sequence "aaa" is not yet defined in this session
select nextval('aaa');
 nextval
---------
      13
(1 row)
```

Установка значения последовательности

- Установка значения последовательности:

```
select setval('aaa', 8, true);
alter sequence aaa restart with 8;
```

- Функция **`setval()`** устанавливает для последовательности заданное значение поля `last_value`
- Если `is_called=true` то, при следующем вызове `nextval()`, последовательность продвинется на следующее значение, которое и будет возвращено функцией `nextval()`
- Если `is_called=false`, то первый вызов `nextval()` вернёт `last_value`, а продвижение последовательности произойдёт только при следующем вызове `nextval()`

Установка значения последовательности

Установить значение последовательности можно функцией:
`select setval(имя, last_value, is_called);`
Функция **`setval()`** устанавливает для последовательности заданное во втором параметре значение для `last_value` и значение из третьего параметра для `is_called`.
Если `is_called=true` то, при следующем вызове `nextval()`, последовательность должна сначала перейти на следующее значение, которое и будет возвращено функцией `nextval()`.
Если `is_called=false`, то первый вызов `nextval()` вернёт `last_value`, а продвижение последовательности произойдёт только при следующем вызове `nextval()`. Пример:
`SELECT setval('aaa', 42, true);` Следующий вызов **`nextval()`** вернёт 42+5=`47`
`SELECT setval('aaa', 42, false);` Следующий вызов **`nextval()`** вернёт `42`
Функция **`nextval()`** не откатывается. Если **`nextval()`** вызывается в транзакции, которая откатывается, то выбранное функцией значение в последовательность не возвращается. Если **`nextval()`** вызывается одновременно в нескольких сессиях, в результате каждого вызова будут гарантированно получены разные значения.
Установить значение последовательности можно и командой:
`alter sequence aaa restart with 8;`
Эта команда устанавливает для последовательности `is_called=false`.

Физическая реплика для обновления

- Пример создания и запуска исходного кластера:

```
export OLD=/opt/tantor/db/17/bin
export NEW=/opt/tantor/db/18/bin
export HOME=/var/lib/postgresql
$OLD/initdb -k -g -D $HOME/tantor-se-17/data
echo "port=5433" >> $HOME/tantor-se-17/data/postgresql.auto.conf
echo "wal_level=logical" >> $HOME/tantor-se-17/data/postgresql.auto.conf
$OLD/pg_ctl start -D $HOME/tantor-se-17/data
```

- создание и запуск физической реплики:

```
$OLD/pg_basebackup -p 5433 -D $HOME/tantor-se-17/data1 -P -R -C --slot=replica1
echo "port=5434" >> $HOME/tantor-se-17/data1/postgresql.auto.conf
$OLD/pg_ctl start -D $HOME/tantor-se-17/data1
psql -p 5433 -c "drop table if exists t; create table t (c bigserial primary key);"
```

- проверка возможности преобразования в клон:

```
$OLD/pg_ctl stop -D $HOME/tantor-se-17/data1
$OLD/pg_createsubscriber -d postgres -D $HOME/tantor-se-17/data1 -P
"port=5433 dbname=template1" -p 5434 -v --dry-run
Done!
```


Физическая реплика для обновления

В примере создаётся кластер 17 версии, который планируется обновить. Существенно то, что нужно установить `wal_level=logical`. Изменение этого параметра до 19 версии требует перезапуска экземпляра.

Дальше создаётся физическая реплика на порту 5434. До и после создания физической реплики можно вносить изменения в данные кластера: создавать, удалять базы данных, любые объекты. В примере создаётся таблица `t`.

Утилита `pg_createsubscriber` **появилась в 17** версии PostgreSQL. Если мигрировать с более старой версии, то настраивать логическую репликацию придётся вручную. При создании подписок командой `CREATE SUBSCRIBER` выполняется перегрузка данных, то может занять долгое время. Если использовать физическую реплику, в которой данные уже есть, то процедура усложняется. В процедуре будет использоваться функция `pg_replication_origin_advance(pg_OIDofSubscription, LSN)`, которая имеется в версиях PostgreSQL 9.5 и более новых. Утилита `pg_createsubscriber` автоматизирует эти действия.

Перед запуском утилиты `pg_createsubscriber` экземпляр физической реплики должен быть остановлен.

Параметр `--dry-run` утилиты `pg_createsubscriber` позволяет предварительно и, по возможности, проверить отсутствие ошибок. При проверке экземпляр реплики запускается и останавливается два раза, создаются и удаляются публикации, подписки, логические слоты.

Директории табличных пространств

- Пути к директориям табличных пространств:

```
select spcname, pg_tablespace_location(oid) location,
pg_tablespace_size(oid) size from pg_tablespace;
  spcname   |                 location                 |   size
------------+------------------------------------------+----------
 pg_default |                                          | 23007781
 pg_global  |                                          |   587144
 u01        | /var/lib/postgresql/tantor-se-17/tbs/u01 |        0
 u02        | /var/lib/postgresql/tantor-se-17/tbs/u02 |        0
```

- Указание новых директорий для табличных пространств при создании физической реплики:

```
$OLD/pg_basebackup -p 5433 -D $HOME/tantor-se-17/data1 -P -R -c fast -C
--slot=replica1 --tablespace-mapping=$HOME/tantor-se-
17/tbs/u01=$HOME/tantor-se-18/tbs/u01 --tablespace-
mapping=$HOME/tantor-se-17/tbs/u02=$HOME/tantor-se-18/tbs/u02
```


Директории табличных пространств

Если в исходном кластере есть табличные пространства, кроме стандартных, то они располагаются вне PGDATA. При создании физической реплики на том же хосте, нужно указать переназначение директорий для этих табличных пространств. Если не указать, то утилита резервирования попытается скопировать файлы в те же самые места и выдаст ошибку о том, то место назначения не пусто. Новые директории нужно создать заранее и директории должны быть пустыми:

```
rm -rf $HOME/tantor-se-17/tbs
mkdir -p -v $HOME/tantor-se-17/tbs/u01
mkdir -p -v $HOME/tantor-se-17/tbs/u02
psql -p 5433 -c "create tablespace u01 location '$HOME/tantor-se-17/tbs/u01'"
psql -p 5433 -c "create tablespace u02 location '$HOME/tantor-se-17/tbs/u02'"
rm -rf $HOME/tantor-se-18/tbs
mkdir -p -v $HOME/tantor-se-18/tbs/u01
mkdir -p -v $HOME/tantor-se-18/tbs/u02
```

Пути к директориям нужно указывать **абсолютные** и в точности такие, на которые указывают символические ссылки в директории PGDATA/pg_tblspc исходного кластера. Точный путь к директориям можно получить функцией `pg_tablespace_location()` на исходном кластере:

```
select spcname, pg_tablespace_location(oid) location, pg_tablespace_size(oid)
size from pg_tablespace;
```

В параметры нужно добавить параметры `--tablespace-mapping=откуда=куда`, по одному параметру для каждого табличного пространства. Все пути должны быть абсолютными и, желательно, находиться вне директорий `PGDATA`.

Запрет DDL команд триггером на события

- До преобразования физической реплики в клон (командой `pg_createsubscriber`) можно создать триггерную функцию и событийный триггер в каждой базе:

```
psql -d postgres -p 5433 -qc "CREATE OR REPLACE FUNCTION
block_ddl() RETURNS event_trigger LANGUAGE plpgsql AS \$\$BEGIN
RAISE EXCEPTION 'DDL-команды заблокированы: идёт миграция';
END\$\$;"
psql -p 5433 -qc "CREATE EVENT TRIGGER trg_block_ddl ON
ddl_command_start EXECUTE FUNCTION block_ddl();"
```

- Событийные триггеры:
  - можно отключить параметром `event_triggers`
  - можно включить/отключить только на реплике или везде
  - появились **в 14 версии** PostgreSQL

Запрет DDL команд триггером на события

Логическая репликация **в 18 версии** не реплицирует DDL команды. Изменения объектов на исходном кластере, созданные таблицы не реплицируются. Хуже того, DDL команда может привести к тому, что подписка не сможет выполнить команду и остановится, что увеличит лаг и потребует устранения расхождения. Для запрета команд DDL, начиная с преобразования физической реплики в клон, можно создать триггерную функцию и триггер **в каждой базе**:

```
CREATE OR REPLACE FUNCTION block_ddl() RETURNS event_trigger
LANGUAGE plpgsql AS
$$
DECLARE obj record;
BEGIN
  FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands() WHERE schema_name
IS NOT NULL
   LOOP
    IF obj.schema_name !~ '^(pg_temp|pg_toast_temp)' THEN
     RAISE EXCEPTION 'DDL-команды заблокированы: идёт миграция';
    END IF;
  END LOOP;
END
$$;
CREATE EVENT TRIGGER trg_block_ddl ON ddl_command_end EXECUTE FUNCTION
block_ddl();
```

Если не использовать функцию `pg_event_trigger_ddl_commands()`, то можно создать триггер на событие `ddl_command_start`.

Функция позволяет создавать, менять и удалять временные объекты, то есть те, которые создаются во временных схемах. Временные объекты не реплицируются. Отключение и удаление триггера:

```
psql -p 5433 -qc "ALTER EVENT TRIGGER trg_block_ddl DISABLE; DROP EVENT
TRIGGER IF EXISTS trg_block_ddl; DROP FUNCTION IF EXISTS block_ddl();"
```

Событийные триггеры можно отключить параметром конфигурации `event_triggers`.

Событийные триггеры появились **в 14 версии** PostgreSQL.

Триггеры при логической репликации

- Значение в столбце `tgenabled` таблиц `pg_event_trigger` и `pg_trigger` отражает, при каких значениях параметра `session_replication_role` срабатывает триггер:
- O = `origin` и `local` (значения равнозначны),
- D = триггер отключён (`disable`),
- R = `replica`,
- A = триггер срабатывает всегда (`always`)
- Значения устанавливаются командами:

```
O = alter event trigger trg_block_ddl enable;
D = alter event trigger trg_block_ddl disable;
R = alter event trigger trg_block_ddl enable replica;
A = alter event trigger trg_block_ddl enable always;
```

- По умолчанию, все триггеры имеют свойство `origin`

Триггеры при логической репликации

По умолчанию, все (событийные, обычные, служебные) триггеры имеют свойство `trgenabled=O`.

Процессы, применяющие изменения на подписчике (logical replication worker) устанавливают для своих сессий `session_replication_role = replica`. Поэтому, триггеры не срабатывают на подписчике. Логическая репликация в PostgreSQL воспринимает значения `origin` и `local` как равнозначные.

Событийные триггеры, обычные триггеры и триггеры для внешних ключей могут быть включены/отключены только для подписчика или везде.

Установка `session_replication_role = replica` отключает все триггеры, в том числе, и служебные, которые реализуют проверки внешних ключей и откладываемые (`DEFERRABLE`) ограничения (`PRIMARY KEY`, `UNIQUE`, `EXCLUDE`). Это позволяет логической репликации менять порядок внесения изменений и не получать ошибки.

Физическая репликация не вызывает срабатывания триггеров, та как работает на физическом уровне, за исключением событийных триггеров типа `login`, которые срабатывают на физических репликах. Важно, чтобы событийный триггер `login`, проверял функцией `pg_is_in_recovery()`, не находится ли кластер в состоянии восстановления перед выполнением команд, которые не могут выполняться на физических репликах. Необработанная ошибка в триггере не позволит создать сессию. Пример:

```
DROP EVENT TRIGGER IF EXISTS init_session;
CREATE OR REPLACE FUNCTION init_session() RETURNS event_trigger LANGUAGE
plpgsql AS
$$
BEGIN
  IF pg_is_in_recovery() THEN
    RAISE NOTICE 'replica';
  ELSE
    RAISE NOTICE 'master';
END IF;
END;
$$;
CREATE EVENT TRIGGER init_session ON login EXECUTE FUNCTION init_session();
ALTER EVENT TRIGGER init_session ENABLE ALWAYS;
```

Служебные триггеры для ограничений целостности

- Внешние ключи и свойство DEFERRABLE реализуются служебными триггерами
- Откладывать проверку могут только UNIQUE, PRIMARY KEY, FOREIGN KEY, EXCLUDE

```
CREATE TABLE t1 (id int PRIMARY KEY DEFERRABLE, id1 INT unique INITIALLY DEFERRED);
CREATE TABLE t (id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY);
CREATE TABLE td (id int PRIMARY KEY REFERENCES t(id));
select tgrelid::regclass tab, tgconstrindid::regclass index, tgfoid::regproc
function_name, tgtype t, tgenabled e, tgisinternal i, tgdeferrable d, tginitdeferred
id from pg_trigger;
 tab |   index     |     function_name      | t  | e | i | d | id
-----+-------------+------------------------+----+---+---+---+---
 t1  | t1_pkey     | unique_key_recheck     | 21 | O | t | t | f
 t1  | t1_id1_pkey | unique_key_recheck     | 21 | O | t | t | t
 t   | t_pkey      | "RI_FKey_noaction_del" |  9 | O | t | f | f
 t   | t_pkey      | "RI_FKey_noaction_upd" | 17 | O | t | f | f
 td  | t_pkey      | "RI_FKey_check_ins"    |  5 | O | t | f | f
 td  | t_pkey      | "RI_FKey_check_upd"    | 17 | O | t | f | f
```


Служебные триггеры для ограничений целостности

Внешние ключи и ограничения целостности с отложенной проверкой в PostgreSQL реализованы служебными триггерами ограничений:

```
DROP TABLE IF EXISTS t, t1, td;
CREATE TABLE t1 (id int PRIMARY KEY DEFERRABLE,
                 id1 INT UNIQUE INITIALLY DEFERRED);
CREATE TABLE t (id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY);
CREATE TABLE td (id int PRIMARY KEY REFERENCES t(id));
select tgrelid::regclass tab, tgconstrindid::regclass index, tgfoid::regproc
function_name, tgtype t, tgenabled e, tgisinternal i, tgdeferrable d,
tginitdeferred id from pg_trigger;
 tab |   index     |     function_name      | t  | e | i | d | id
-----+-------------+------------------------+----+---+---+---+---
 t1  | t1_pkey     | unique_key_recheck     | 21 | O | t | t | f
 t1  | t1_id1_pkey | unique_key_recheck     | 21 | O | t | t | t
 t   | t_pkey      | "RI_FKey_noaction_del" |  9 | O | t | f | f
 t   | t_pkey      | "RI_FKey_noaction_upd" | 17 | O | t | f | f
 td  | t_pkey      | "RI_FKey_check_ins"    |  5 | O | t | f | f
 td  | t_pkey      | "RI_FKey_check_upd"    | 17 | O | t | f | f
```

При создании внешнего ключа создаются четыре триггера с условиями срабатывания:

AFTER INSERT и AFTER UPDATE FOR EACH ROW триггеры на дочернюю таблицу, которые **проверяют, что строка дочерней таблицы ссылается на строку родительской.**

AFTER UPDATE и AFTER DELETE FOR EACH ROW триггеры на родительскую таблицу, которые проверяют, что UPDATE и DELETE **не породили осиротевших строк в дочерней таблице**.

Условия срабатывания хранятся в столбце **tgtype** в виде битовой карты.

Откладывать проверку до фиксации транзакции (DEFERRABLE) могут только ограничения UNIQUE, PRIMARY KEY, **FOREIGN KEY**, EXCLUDE. Ограничения NOT NULL и CHECK не могут откладывать проверку.

Триггеры ограничений ("*constraint triggers*") создаются командой:
CREATE **CONSTRAINT** TRIGGER имя AFTER .. FOR EACH ROW.

Триггеры ограничений аналогичны обычным триггерам, только момент их срабатывания может настраиваться командой внутри транзакции командой SET CONSTRAINTS {ALL | имя_ограничения} [IMMEDIATE | **DEFERRED**].

Статус VALIDATED у DEFERRABLE ограничений

- DEFERRABLE не позволяет планировщику предполагать уникальность и использовать оптимизации
- Не стоит полагаться на свойство **VALIDATED** у DEFERRABLE ограничений целостности

```
create table t (id int, unique (id) DEFERRABLE INITIALLY IMMEDIATE);
set session_replication_role = replica;
insert into t values (1);
insert into t values (1);
select * from t;
 id
----
  1
  1
select conrelid::regclass table, conname, contype, condeferrable,
convalidated FROM pg_constraint WHERE conname='t_id_key';
 table | conname  | contype | condeferrable | convalidated
-------+----------+---------+---------------+--------------
 t     | t_id_key | u       | t             | t
```


Статус VALIDATED у DEFERRABLE ограничений

Если отключить срабатывание триггеров (это можно сделать параметром `session_replication_role = replica`), то ограничение целостности не будет срабатывать и можно будет вставить строки, нарушающие ограничение:

```
create table t (id int, unique (id) DEFERRABLE INITIALLY IMMEDIATE);
set session_replication_role = replica;
insert into t values (1);
insert into t values (1);
```

В таблице будет две одинаковые (неуникальные) строки. При этом, ограничение целостности будет иметь свойство **VALIDATED**. Не стоит полагаться на свойство **VALIDATED** у DEFERRABLE ограничений целостности, **VALIDATED** не гарантирует отсутствие нарушений ограничения целостности существующими строками.

```
select conrelid::regclass table, conname, contype, condeferrable,
convalidated FROM pg_constraint WHERE conname='t_id_key';
 table | conname  | contype | condeferrable | convalidated
-------+----------+---------+---------------+--------------
 t     | t_id_key | u       | t             | t
```

Если ограничение UNIQUE, PRIMARY KEY, FOREIGN KEY, EXCLUDE не откладываемое, то триггер не создаётся, так как проверка уникальности проверяется немедленно. Немедленная проверка позволяет планировщику использовать оптимизации, полагающиеся на предположения об уникальности. Пример запроса, план и время выполнения которого будут отличаться для DEFERRABLE и NOT DEFERRABLE ограничения:

```
EXPLAIN (analyze, timing off) SELECT * FROM t WHERE id IN (SELECT id FROM t);
```

будет использоваться Semi Join двух Seq Scan.

Для NOT DEFERRABLE:

```
truncate t; ALTER TABLE t DROP CONSTRAINT t_id_key, ADD CONSTRAINT t_id_key
UNIQUE (id) NOT DEFERRABLE;
```

будет использоваться Seq Scan:

```
Seq Scan on t  (cost=0.00..35.50 rows=2537 width=4) (actual rows=0 loops=1)
   Filter: (id IS NOT NULL)
```

https://habr.com/ru/companies/tantor/articles/1055860/

Функция pg_get_triggerdef(oid)

- Возвращает определение триггера
- В определении (команде создания) есть условия, по которым срабатывает триггер и тип триггера
- RI_FKey_check_ins() и RI_FKey_check_upd() на INSERT и UPDATE на дочерней таблице
- RI_FKey_{cascade|restrict|setnull|setdefault|noaction}*_{del | upd}() на DELETE и UPDATE на родительской таблице

```
select tgrelid::regclass tab, tgconstrindid::regclass index,
tgfoid::regproc as function_name, tgenabled e, tgisinternal i,
pg_get_triggerdef(oid) from pg_trigger limit 1;
 tab |  index  |   function_name    | e | i | pg_get_triggerdef
-----+---------+--------------------+---+---+-------------------
  t1 | t1_pkey | unique_key_recheck | O | t | CREATE CONSTRAINT TRIGGER
"PK_ConstraintTrigger_16733" AFTER INSERT OR UPDATE ON public.t1 DEFERRABLE
INITIALLY DEFERRED FOR EACH ROW EXECUTE FUNCTION unique_key_recheck()
```


Функция pg_get_triggerdef(oid)

Функция pg_get_triggerdef(oid) возвращает определение триггера. Пример:

```
select tgrelid::regclass tab, tgconstrindid::regclass index,  tgfoid::regproc
as function_name, tgenabled e, tgisinternal i, pg_get_triggerdef(oid) from
pg_trigger limit 1;
 tab |  index  |   function_name    | e | i | pg_get_triggerdef
-----+---------+--------------------+---+---+-------------------
  t1 | t1_pkey | unique_key_recheck | O | t | CREATE CONSTRAINT TRIGGER
"PK_ConstraintTrigger_16733" AFTER INSERT OR UPDATE ON public.t1 DEFERRABLE
INITIALLY DEFERRED FOR EACH ROW EXECUTE FUNCTION unique_key_recheck()
```

В определении (команде создания) есть условия, по которым срабатывает триггер и тип триггера. Тип триггера можно также получить из столбца tgtype запросом:

```
select tgrelid::regclass tab, tgconstrindid::regclass index, tgname,
tgfoid::regproc as function_name, CASE tgtype & 66 WHEN 2 THEN 'BEFORE ' WHEN
64 THEN 'INSTEAD OF ' ELSE 'AFTER ' END || CASE tgtype & 60 WHEN 4 THEN
'INSERT' WHEN 8 THEN 'DELETE' WHEN 16 THEN 'UPDATE' WHEN 20 THEN 'INSERT
UPDATE' WHEN 32 THEN 'TRUNCATE' ELSE '?' END || ' FOR EACH ' || CASE WHEN
tgtype & 1 = 1 THEN 'ROW' ELSE 'STATEMENT' END AS firing_conditions, tgenabled
e, tgisinternal i, tgdeferrable d, tginitdeferred id from pg_trigger;
 tab |   index    |         firing_conditions          | e | i | d | id
-----+------------+------------------------------------+---+---+---+---
 t1  | t1_pkey    | AFTER INSERT UPDATE FOR EACH ROW | O | t | t | t
```

RI_FKey_check_ins() и RI_FKey_check_upd() срабатывают при INSERT и UPDATE на дочерней таблице и проверяют наличие строк в родительской. Остальные RI_FKey_{cascade | restrict | setnull | setdefault | noaction}*_{del | upd}() срабатывают при DELETE и UPDATE на родительской таблице и используются при каскадном удалении и изменении строк NO ACTION | RESTRICT | CASCADE | SET NULL.

Названия функций, обслуживающих служебные триггеры, можно получить запросом:

```
SELECT oid, proname FROM pg_proc where proname like '%RI%' OR proname like
'uniq%';
  oid  |         proname
------+-------------------------
 1250 | unique_key_recheck
```

Внешние ключи и блокировки

- При наличии внешнего ключа, в таблицу, где он создан блокируются вставки строк, если на родительской таблице заблокирована строка, на которую ссылается вставляемая строка командами:
  - `DELETE`
  - `UPDATE` который меняет ключевой столбец
  - `SELECT ... FOR UPDATE`
- `SELECT ... FOR NO KEY UPDATE` не блокирует вставку
- Если удалять дочернюю таблицу с внешним ключом или удалить внешний ключ, то на родительскую таблицу будет установлена блокировка `ACCESS EXCLUSIVE`
- ожидание получения `ACCESS EXCLUSIVE` блокирует все запросы к таблице

Внешние ключи и блокировки

При наличии внешних ключей при выполнении обновления столбца, на который ссылается внешний ключ, удалении строки или `SELECT FOR UPDATE`, блокируется добавление новых строк в дочернюю таблицу со значениями, на которые ссылается внешний ключ. Транзакции с `INSERT` в дочернюю таблицу будут ждать, что создаёт узкое место.

Рекомендация: использовать `SELECT FOR NO KEY UPDATE` на строки родительской таблицы, он позволяет выполнять `INSERT` в дочернюю таблицу.

Стоит проверять код и там, где используется `SELECT FOR UPDATE`, но не меняется ключевой столбец, заменить `SELECT FOR UPDATE` на `SELECT FOR NO KEY UPDATE`.

Если меняется значение в столбце, на который ссылается внешний ключ или трока удаляется, то блокирование `INSERT` в дочерней таблице ожидаемо, чтобы защититься от появления осиротевших строк. `SELECT FOR UPDATE` можно использовать только когда планируется делать `DELETE` этой строки или `UPDATE` в полях первичного (или уникального) ключа, на которые могут ссылаться внешние ключи.

`FOR UPDATE` устанавливает на строки блокировку, которая конфликтует со всеми остальными типами `SELECT FOR ..`

`FOR NO KEY UPDATE` совместим с `FOR KEY SHARE`. Когда `INSERT` вставляет строку в дочернюю таблицу, идёт попытка получить блокировку строки родительской таблицы (на которую ссылается вставляемая строка) в режиме `FOR KEY SHARE`.

Если удалять дочернюю таблицу с внешним ключом или удалить внешний ключ, то на родительскую таблицу будет установлена блокировка `ACCESS EXCLUSIVE`. Если есть долгие запросы к родительской таблице, то удаление триггера встаёт в очередь за за командами, которые удерживают блокировки на родительскую таблицу и блокирует все запросы к родительской таблице, так как `ACCESS EXCLUSIVE` не позволяет работать даже `SELECT`.

Это происходит из-за того, что внешний ключ использует два системных триггера на родительской таблице, а удаление триггеров требует `ACCESS EXCLUSIVE` на той таблице, на которой созданы триггеры.

https://habr.com/ru/companies/tantor/articles/940066/

Создание клона из физической реплики

- преобразование в клон:

```
$OLD/pg_createsubscriber -d postgres -D $HOME/tantor-se-17/data1
-P "port=5433 dbname=template1" -p 5434 -v
```

- создание пустого кластера для обновления клона:

```
$NEW/pg_ctl stop -D $HOME/tantor-se-18/data1
rm -rf $HOME/tantor-se-18/data1
$NEW/initdb -k -g -D $HOME/tantor-se-18/data1
```

- обновление клона на новую версию и проверки:

```
$NEW/pg_upgrade --link -b $OLD -B $NEW -d $HOME/tantor-se-17/data1
-D $HOME/tantor-se-18/data1
echo "port=5434" >> $HOME/tantor-se-18/data1/postgresql.auto.conf
$NEW/pg_ctl start -D $HOME/tantor-se-18/data1
```

```
psql -p 5433 -qc "insert into t values (default) returning *"
psql -p 5434 -qc "select max(c) from t"
psql -p 5433 -qc "select slot_name, active, synced,
pg_current_wal_lsn()-confirmed_flush_lsn lag_bytes,
confirmed_flush_lsn, inactive_since from pg_replication_slots"
```


Создание клона из физической реплики

Утилиту `pg_createsubscriber` нужно запускать **на той же версии, что и кластер**. В примере обновляется 17 версия и утилита запускается из директории 17 версии.

В 18 версии появился параметр `-a` (`--all`), который удобно использовать вместо параметра `-d`. Параметр создаёт по одной подписке-публикации на каждую базу, кроме шаблонных и тех, к которым запрещено подключение. В 17 версии вместо параметра моно указать несколько параметров `-d`, после которых указать все базы данных кластера.

Можно не запускать экземпляр клона, а сразу перейти к его обновлению на новую версию утилитой `pg_upgrade`. Также можно обновить утилитой `pg_dumpall`. Если нужна отказоустойчивость даже на время обновления, то на обновленный клон можно создать физическую реплику.

После обновления нужно запустить экземпляр клона и дождаться передачи и применения (**стабильно небольшой лаг**) всех накопившихся изменений **по всем** парам публикация-подписка. У каждой подписки свой лаг, который зависит от активности приложений, объема данных в транзакциях и числа транзакций.

Проверять отставание (лаг) можно через представление `pg_replication_slots` на исходном (публикующем)кластере по всем слотам логической репликации:

```
psql -p 5433 -qc "select slot_name, active, synced, pg_current_wal_lsn()-
confirmed_flush_lsn lag_bytes, confirmed_flush_lsn, inactive_since from
pg_replication_slots"
```

Отставание можно проверять и по другому представлению:

```
psql -p 5433 -qc "select application_name, state, replay_lsn, sent_lsn-
replay_lsn lag_stat from pg_stat_replication"
```

До версии 18.1 (и 17.7) у представления `pg_stat_replication` была ошибка: если при воспроизведении WAL на любом подписчике (реплике) LSN переставал увеличиваться, то обновление столбцов `write_lag` и `flush_lag` прекращалось.
(https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=62d5ee75b)

Установка значений последовательностей

- удаление старой версии клона:

```
rm -rf $HOME/tantor-se-17/data1
```

- Остановить на исходном кластере фоновые задачи, которые могут вносить изменения в объекты
- Разорвать сессии и запретить приложениям подсоединяться к исходному кластеру. Пример:

```
$OLD/pg_ctl -D $HOME/tantor-se-17/data restart -o "-c
listen_addresses='' -c unix_socket_permissions=0700"
```

- переустановить значения последовательностей во всех базах кластера. Пример для одной базы:

```
psql -d postgres -p 5433 -qtc "select format('select setval(''%I.%I'',
%s, true);', schemaname , sequencename, last_value) c from pg_sequences
where last_value is not null;" | psql -d postgres -p 5434
```


Установка значений последовательностей

Старую версию клона можно удалить, чтобы освободить место.

Остановить на исходном кластере фоновые задачи, которые могут вносить изменения в таблицы исходного кластера.

Дождавшись минимального отставания всех подписок нового кластера от исходного кластера, остановить обслуживание клиентов исходным кластером и запретить к нему подсоединяться. Для этого можно перезапустить экземпляр исходного кластера с запретом сетевых подсоединений:

```
$OLD/pg_ctl -D $HOME/tantor-se-17/data restart -o "-c listen_addresses='' -c
unix_socket_permissions=0700"
```

Также можно запретить соединения приложений к базам данных в файле `pg_hba.conf`

Можно было бы создать файл `standby.signal` в директории исходного кластера:

```
touch $HOME/tantor-se-17/data/standby.signal
```

и перезапустить экземпляр, но подписчики могут подсоединяться к физической реплике только с 17 версии. Это имело бы смысл для того, чтобы, при наличии лага, процессы `walsender` исходного кластера могли бы передать изменения для устранения лага. Но, в любом, случае, нужно было бы проверить, что лаг по всем парам публикация-подписка нулевой. Также на физических репликах нельзя использовать функцию `nextval()`.

Переустановить значения последовательностей во всех базах данных. В примере переустанавливаются последовательности, которые использовались хоть один раз, базы данных `postgres`:

```
psql -d postgres -p 5433 -qtc "select format('select setval(''%I.%I'', %s,
true);', schemaname , sequencename, last_value) c from pg_sequences where
last_value is not null;" | psql -d postgres -p 5434
```

Последовательности, из которых ни разу не выбирались значения, выгружать не нужно, поэтому используется условие `last_value is not null`.

Пример аналогичного запроса:

```
psql -d postgres -p 5433 -qtc "select format('select setval(''%I.%I'', %s,
true);', n.nspname, c.relname, pg_sequence_last_value(n.nspname || '.' ||
c.relname)) FROM pg_class c JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind = 'S' AND pg_sequence_last_value(n.nspname || '.' ||
c.relname) is not null;" | psql -d postgres -p 5434
```

Переключение на новый кластер

- В 18 версии появилась функция `pg_get_sequence_data` и можно её можно использовать при миграции с 18 версии вместо функции `nextval`
- В 19 версии последовательности можно обновить командой `ALTER SUBSCRIPTION имя REFRESH SEQUENCES`
- дождаться нулевого лага по всем слотам репликации:

```
psql -p 5433 -qc "select slot_name, active, synced,
pg_current_wal_lsn()-confirmed_flush_lsn lag_bytes,
confirmed_flush_lsn, inactive_since from pg_replication_slots"
psql -p 5433 -qc "select application_name, state, replay_lsn,
sent_lsn-replay_lsn lag_stat from pg_stat_replication"
```

- Разрешить приложениям подсоединяться к новому кластеру

Переключение на новый кластер

В 18 версии появилась функция `pg_get_sequence_data` и можно её можно использовать при миграции с 18 версии вместо функции `nextval`:

```
select 'select setval(''' || seq_name || ''', ' || last_value || ', ' ||
is_called || ');' from (select schemaname || '.' || sequencename seq_name,
(select last_value from pg_get_sequence_data(schemaname || '.' ||
sequencename)) , (select is_called from pg_get_sequence_data(schemaname || '.'
|| sequencename)) from pg_sequences);
```

В 19 версии последовательности можно обновить командой `ALTER SUBSCRIPTION имя REFRESH SEQUENCES`., это нужно сделать до удаления подписок, исходный кластер должен работать и не обслуживать приложения.

После перегрузки значений последовательностей, дождаться передачи и применения (лаг должен быть нулевым) всех накопившихся изменений по всем парам публикация-подписка.

Разрешить приложениям подсоединяться к новому кластеру.

После переключения

- удалить подписки во всех базах обновлённого кластера:

```
psql -p 5434 -qtc "select 'alter subscription ' || subname || ' disable;' from
(select subname from pg_subscription)" | psql -p 5434
psql -p 5434 -qtc "select 'alter subscription ' || subname || ' set (slot_name =
none);' from (select subname from pg_subscription)" | psql -p 5434
psql -p 5434 -qtc "select 'drop subscription ' || subname || ';' from (select
subname from pg_subscription)" | psql -p 5434
```

- удалить публикации и слоты репликации на всех базах исходного кластера:

```
psql -d postgres -p 5433 -qtc "select 'select pg_drop_replication_slot(''' ||
slot_name || ''');' from (select slot_name from pg_replication_slots)" | psql -d
postgres -p 5435
psql -d postgres -p 5433 -qc "ALTER EVENT TRIGGER trg_block_ddl DISABLE; DROP
EVENT TRIGGER IF EXISTS trg_block_ddl; DROP FUNCTION IF EXISTS block_ddl();"
psql -d postgres -p 5433 -qtc "select 'drop publication ' || pubname || ';' from
(select pubname from pg_publication)" | psql -d postgres -p 5433
```


После переключения

Удалить подписки во всех базах данных нового кластера:

```
psql -p 5434 -qc "select max(*) from t"
psql -p 5434 -qtc "select 'alter subscription ' || subname || ' disable;'
from (select subname from pg_subscription)" | psql -p 5434
psql -p 5434 -qtc "select 'alter subscription ' || subname || ' set
(slot_name = none);' from (select subname from pg_subscription)" | psql -p
5434
psql -p 5434 -qtc "select 'drop subscription ' || subname || ';' from (select
subname from pg_subscription)" | psql -p 5434
```

Если исходным кластером планируется пользоваться или оставить его на всякий случай, то можно удалить слоты репликации и подписки во всех его базах данных. Удалить событийный триггер, так как он не даст удалить публикацию командой `drop publication`.

Пример команд для одной базы данных с названием `postgres`:

```
psql -d postgres -p 5433 -qtc "select 'select pg_drop_replication_slot(''' ||
slot_name || ''');' from (select slot_name from pg_replication_slots)" | psql
-d postgres -p 5435
psql -d postgres -p 5433 -qc "ALTER EVENT TRIGGER trg_block_ddl DISABLE; DROP
EVENT TRIGGER IF EXISTS trg_block_ddl; DROP FUNCTION IF EXISTS block_ddl();"
psql -d postgres -p 5433 -qtc "select 'drop publication ' || pubname || ';'
from (select pubname from pg_publication)" | psql -d postgres -p 5433
```

Если материализованные представления обновлялись на исходном кластере, пока шла миграция, то обновить их и на новом кластере.

Если использовались большие объекты (lo), то сравнить их и перегрузить из исходного кластера большие объекты, если есть расхождения.

После того, как старый кластер станет не нужен, остановить экземпляр исходного кластера и удалить его, чтобы освободить место:

```
$OLD/pg_ctl -D $HOME/tantor-se-17/data stop
rm -rf $HOME/tantor-se-17/data
```