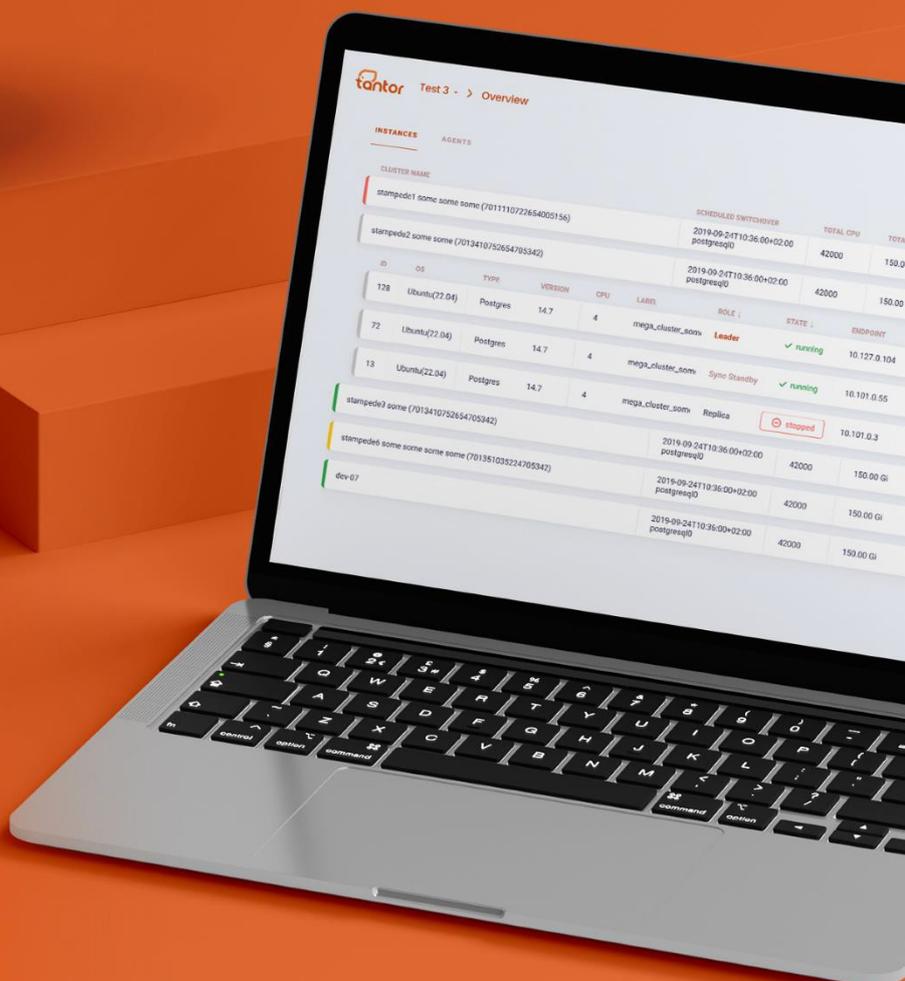


DBA1-17 Tantor: Администрирование PostgreSQL 17

Практические задания



Автор: Олег Иванов

Оглавление

глава	страница
Глава 1. Установка Tantor Postgres	3
Глава 2a. Архитектура	32
Глава 2b. Многоверсионность	37
Глава 2c. Регламентные работы	44
Глава 2d. Выполнение запросов	51
Глава 2e. Расширения	55
Глава 3. Конфигурирование	61
Глава 4a. Логическая структура кластера	79
Глава 4b. Физическая структура кластера	90
Глава 5. Журналирование	114
Глава 6. Безопасность	117
Глава 7a. Физическое резервирование	126
Глава 7b. Логическое резервирование	143
Глава 8a. Физическая репликация	150
Глава 8b. Логическая репликация	183
Глава 10. Новые возможности Tantor Postgres	201

Авторские права

Учебное пособие, практические задания, презентации (далее документы) предназначены для учебных целей.

Документы защищены авторским правом и законодательством об интеллектуальной собственности.

Вы можете копировать и распечатывать документы для личного использования в целях самообучения, а также при обучении в авторизованных ООО "Tantor Labs" учебных центрах и образовательных учреждениях. Авторизованные ООО "Tantor Labs" учебные центры и образовательные учреждения могут создавать учебные курсы на основе документов и использовать документы в учебных программах с письменного разрешения ООО "Tantor Labs".

Вы не имеете права использовать документы для платного обучения сотрудников или других лиц без разрешения ООО "Tantor Labs". Вы не имеете права лицензировать, коммерчески использовать документы полностью или частично без разрешения ООО "Tantor Labs".

При некоммерческом использовании (презентации, доклады, статьи, книги) информации из документов (текст, изображения, команды) сохраняйте ссылку на документы.

Текст документов не может быть изменен каким-либо образом.

Информация, содержащаяся в документах, может быть изменена без предварительного уведомления и мы не гарантируем ее безошибочность. Если вы обнаружите ошибки, нарушение авторских прав, пожалуйста, сообщите нам об этом.

Отказ от ответственности за содержание документа, продукты и услуги третьих лиц:

ООО "Tantor Labs" и связанные лица не несут ответственности и прямо отказываются от любых гарантий любого рода, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием документа. ООО "Tantor Labs" и связанные лица не несут ответственности за любые убытки, издержки или ущерб, возникшие в результате использования информации, содержащейся в документе или использования сторонних ссылок, продуктов или услуг.

Авторское право © 2025, ООО "Tantor Labs"

Автор: Олег Иванов

Создан: 13 июля 2025 г.

По вопросам обучения обращайтесь: edu@tantorlabs.ru

Глава 1. Установка Tantor Postgres

Часть 1. Создание кластера

1) Откройте терминал с правами root:

```
astra@tantor:~$ sudo bash
```

2) Посмотрите сколько ядер процессора доступно в виртуальной машине (результат может отличаться от приведенных как пример значений):

```
root@tantor:/home/astra# cat /proc/cpuinfo | grep cores
cpu cores      : 2
cpu cores      : 2
```

Число строк по числу процессоров. Если выполнить команду без "`| grep cores`" будет видно, что выдаются детальные данные по каждому ядру процессора.

Сколько оперативной памяти имеется:

```
root@tantor:/home/astra# cat /proc/meminfo | grep Mem
MemTotal:      2981180 kB
MemFree:       1306840 kB
MemAvailable:  2168596 kB
```

3) Программное обеспечение Tantor Postgres установлено в директорию `/opt/tantor/db`

Директория с файлами кластера: `/var/lib/postgresql`

Эти директории могут иметь отдельные точки монтирования, но в нашей операционной системе эти директории смонтированы в корне `"/`. Проверьте сколько осталось свободного места:

```
root@tantor:/home/astra# df -HT | grep /$
/dev/sda1      ext4    50G    17G    31G   36% /
```

Свободны 31 ГБ.

При промышленной эксплуатации рекомендуется иметь 4 ядра.

Оперативной памяти: по крайней мере 4 ГБ.

Свободного места на системе хранения ("диске"): 40 ГБ.

4) Скачайте инсталлятор:

```
root@tantor:/home/astra# wget https://public.tantorlabs.ru/db_installer.sh
https://public.tantorlabs.ru/db_installer.sh
Resolving public.tantorlabs.ru (public.tantorlabs.ru)... 84.201.157.208
Connecting to public.tantorlabs.ru (public.tantorlabs.ru)|84.201.157.208|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 18312 (18K) [application/octet-stream]
Saving to: 'db_installer.sh'
db_installer.sh      100%[=====>] 17,88K  -
-.-KB/s in 0s
'db_installer.sh' saved [18312/18312]
```

5) Посмотрите разрешения на исполнение скрипта инсталляции:

```
root@tantor:/home/astra# ls -al db_installer.sh
```

```
-rw-r--r-- 1 root root 18353 db_installer.sh
```

6) Если разрешений на исполнение файла нет, то дайте права на исполнение:

```
root@tantor:/home/astra# chmod +x db_installer.sh
```

7) Проверьте версию инсталлятора и ознакомьтесь с параметрами:

```
root@tantor:/home/astra# ./db_installer.sh --help
=====
Usage: db_installer.sh [OPTIONS]
Installer version: 25.01.29
This script will perform installation of the Tantor DB on current host.
If the Tantor DB is already installed, no actions will be taken.
Available options:
  --help                Show this help message.

-----
  --edition=            Set edition (be, se, se-1c, se-certified). "se" is
default.
  --major-version=     Set major version (14, 15)
  --maintenance-version= Set maintenance version (15.2.4).
                        By default latest version will be installed.
  --do-initdb          After installation run initdb with checksums.
  --package=           Set specific package (all, client, libpq5).
                        "all" is default.

-----
  --from-file=         Install package from local file (rpm, deb)
                        May be used with --do-initdb option

=====
Example for commercial use
=====
  export NEXUS_USER="user_name"
  export NEXUS_USER_PASSWORD="user_password"
  export NEXUS_URL="nexus.tantorlabs.ru"
  ./db_installer.sh \
    --do-initdb \
    --major-version=15 \
    --edition=se
=====
Example for evaluation use (without login and password)
Only for Basic Edition
=====
  export NEXUS_URL="nexus-public.tantorlabs.ru"
  ./db_installer.sh \
    --do-initdb \
    --major-version=15 \
    --edition=be
=====
Examples how to install from file
=====
  ./db_installer.sh \
    --from-file=./packages/tantor-be-server-15_15.4.1.jammy_amd64.deb
  ./db_installer.sh \
    --do-initdb \ --from-file=/tmp/tantor-be-server-15_15.4.1.jammy_amd64.deb
```

При создании кластера инсталлятором включается подсчет контрольных сумм для блоков данных.

8) Переустановите пароль пользователю postgres. Используйте пароль postgres :

```
root@tantor:/home/astra# passwd postgres
New password: postgres
Retype new password: postgres
```

```
passwd: password updated successfully
```

9) Проверьте, что путь к исполняемым файлам был добавлен в файл профиля пользователя `postgres`. Переключитесь в пользователя `postgres`, который создаётся инсталлятором для запуска экземпляров кластеров. Параметр `"-"` заставляет исполнить файлы профилей того пользователя, в которого идёт переключение.

```
root@tantor:/home/astra# su - postgres
postgres@tantor:~$ cat .bash_profile
export PATH=/opt/tantor/db/17/bin:$PATH
export PGDATA=/var/lib/postgresql/tantor-se-17/data
#export LC_MESSAGES=ru_RU.utf8
#unset LANGUAGE
```

10) Этот пункт выполните только, если в файле `.bash_profile` отсутствует переменная окружения `PGDATA`.

Если переменная отсутствует, то добавьте путь к файлам кластера в переменную окружения, чтобы в будущем каждый раз не указывать ее параметром с названием `"-D"` утилитам. Команду вводить одной строкой, нужно использовать две угловые скобки:

```
postgres@tantor:~$
echo "export PGDATA=/var/lib/postgresql/tantor-se-17/data" >> .bash_profile
```

Проверьте, что успешно и правильно добавили `PGDATA` в конец файла профиля.

```
postgres@tantor:~$ cat .bash_profile
export PATH=/opt/tantor/db/17/bin:$PATH
#export LC_MESSAGES=ru_RU.utf8
#unset LANGUAGE
export PGDATA=/var/lib/postgresql/tantor-se-17/data
```

Перечитайте файл профиля, который изменили:

```
postgres@tantor:~$ source .bash_profile
```

Часть 2. Создание кластера утилитой initdb

1) Остановите два экземпляра кластера. Используйте утилиту `pg_ctl`:

```
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$
/usr/lib/postgresql/15/bin/pg_ctl stop -D /var/lib/postgresql/15/main
```

```
waiting for server to shut down.... done
server stopped
root@tantor:~# sudo systemctl stop postgresql
sudo systemctl disable postgresql
```

```
Synchronizing state of postgresql.service with SysV service script with
/lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install disable postgresql
Removed "/etc/systemd/system/multi-user.target.wants/postgresql.service".
```

Вы остановили экземпляр Astralinux PostgreSQL 15. Команда остановки службы `systemctl stop postgresql` не выдала результат, при том что экземпляр уже был остановлен. Можно было использовать команду управления службами, запускаемыми инфраструктурой `systemd`: `sudo systemctl stop tantor-se-server-17`, но гарантий того, что после возврата промпта все процессы остановлены нет.

При запуске командой `systemctl` сначала проверяется, что директория `PGDATA` "похожа" на директорию кластера утилитой `postgresql-check-db-dir`, и потом используется `pg_ctl start`.

2) Откройте текстовым редактором с которым знакомы (`kate` или `mcedit`) файл `/usr/lib/systemd/system/tantor-se-server-17.service`, или, если не знакомы, то командой `cat` и найдите строки, найдите строки, где указаны утилиты, которые вызываются при запуске, остановке или обновлении (`reload`) службы:

```
postgres@tantor:~$
cat /usr/lib/systemd/system/tantor-se-server-17.service | grep /opt

ExecStartPre=/opt/tantor/db/17/bin/postgresql-check-db-dir ${PGDATA}
ExecStart=/opt/tantor/db/17/bin/pg_ctl start -D ${PGDATA} -s -w -t
${PGSTARTTIMEOUT}
ExecStop=/opt/tantor/db/17/bin/pg_ctl stop -D ${PGDATA} -s -m fast
ExecReload=/opt/tantor/db/17/bin/pg_ctl reload -D ${PGDATA} -s
```

Для запуска, остановки, перечитывания конфигурации используется утилита `pg_ctl`.

Обновление (`reload`) - это не перезагрузка (`restart`)

Режим остановки по умолчанию - `fast`.

Если экземпляр запускался утилитой `pg_ctl`, а не через `systemd`, то `systemctl` не остановит экземпляр. При этом, `pg_ctl` останавливает экземпляр, запущенный любым способом. Поэтому рекомендуется останавливать экземпляр утилитой `pg_ctl`.

Запускать экземпляр лучше через `systemctl`. При запуске экземпляра через сетевое соединение (подсоединившись по `ssh`) утилитой `pg_ctl`, **экземпляр принудительно остановится после закрытия сетевого соединения**, так как по умолчанию `KillUserProcesses=yes`, но его можно поменять в `/etc/systemd/logind.conf`. Также, при запуске через `pg_ctl`, нужно настроить вывод журнала сообщений в файл, а не на экран терминала.

3) Выполните ещё раз команду остановки экземпляра. Если экземпляр запущен - он становится, если не запущен - утилита об этом сообщит:

```
postgres@tantor:~$ pg_ctl stop
pg_ctl: PID file "/var/lib/postgresql/tantor-se-17/data/postmaster.pid" does not exist
Is server running?
```

4) Сохраните директорию кластера выполнив три комады:

```
postgres@tantor:~$ mkdir $PGDATA/../data.SAVE
mv $PGDATA/* $PGDATA/../data.SAVE
chmod 750 $PGDATA/../data.SAVE
```

5) Создадим новый кластер. Для создания кластера используется утилита `initdb`. Утилите передаются параметры и она реагирует на переменные окружения, в частности, относящиеся к локализации (но не только). Запустите утилиту без параметров (со значениями по умолчанию):

```
postgres@tantor:~$ initdb
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".
Data page checksums are disabled.
fixing permissions on existing directory /var/lib/postgresql/tantor-se-17/data
... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Europe/Moscow
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok
initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A,
or --auth-local and --auth-host, the next time you run initdb.
Success. You can now start the database server using:
pg_ctl -D /var/lib/postgresql/tantor-se-17/data -l logfile start
```

6) Прочитайте результат. Для этого можно использовать клавиши на клавиатуре `<Shift+PgUp>` `<Shift+PgDown>`. Обратите внимание на то, что до 18 версии PostgreSQL подсчет контрольных сумм, по умолчанию, **не включается**.

Также выдаются параметры локализации, с которыми создан кластер.

7) Проверьте утилитой `pg_controldata`, что подсчет контрольных сумм не включён:

```
postgres@tantor:~$ pg_controldata
pg_control version number:          1300
Catalog version number:            202307071
```

```

Database system identifier: 7340951136757174317
Database cluster state:    shut down
pg_control last modified: 12:19:38
Latest checkpoint location: 0/1514AB0
Latest checkpoint's REDO location: 0/1514AB0
Latest checkpoint's REDO WAL file: 00000001000000000000000001
Latest checkpoint's TimeLineID: 1
Latest checkpoint's PrevTimeLineID: 1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID: 731
Latest checkpoint's NextOID: 13602
Latest checkpoint's NextMultiXactId: 1
Latest checkpoint's NextMultiOffset: 0
Latest checkpoint's oldestXID: 723
Latest checkpoint's oldestXID's DB: 1
Latest checkpoint's oldestActiveXID: 0
Latest checkpoint's oldestMultiXid: 1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid: 0
Latest checkpoint's newestCommitTsXid: 0
Time of latest checkpoint: 12:19:38
Fake LSN counter for unlogged rels: 0/3E8
Minimum recovery ending location: 0/0
Min recovery ending loc's timeline: 0
Backup start location: 0/0
Backup end location: 0/0
End-of-backup record required: no
wal_level setting: replica
wal_log_hints setting: off
max_connections setting: 100
max_worker_processes setting: 8
max_wal_senders setting: 10
max_prepared_xacts setting: 0
max_locks_per_xact setting: 64
track_commit_timestamp setting: off
Maximum data alignment: 8
Database block size: 8192
Blocks per segment of large relation: 131072
WAL block size: 8192
Bytes per WAL segment: 16777216
Maximum length of identifiers: 64
Maximum columns in an index: 32
Maximum size of a TOAST chunk: 1996
Size of a large-object chunk: 2048
Date/time type storage: 64-bit integers
Float8 argument passing: by value
Data page checksum version: 0
Mock authentication nonce:
0d18c599c7876e965a894cd059b60c1307f5e1a959703351495b0193f729174a
    
```

8) Найдите в результате информацию о том, что экземпляр кластера не был запущен или корректно погашен. Это строка:

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

9) Посмотрите параметры командной строки утилиты `pg_checksums`:

```

postgres@tantor:~$ pg_checksums --help
pg_checksums enables, disables, or verifies data checksums in a PostgreSQL
database cluster.
Usage:
  pg_checksums [OPTION]... [DATADIR]
    
```

Options:

```
[-D, --pgdata=]DATADIR data directory
-c, --check check data checksums (default)
-d, --disable disable data checksums
-e, --enable enable data checksums
-f, --filenode=FILENODE check only relation with specified filenode
-N, --no-sync do not wait for changes to be written safely to disk
-P, --progress show progress information
-v, --verbose output verbose messages
-V, --version output version information, then exit
-?, --help show this help, then exit
```

If no data directory (DATADIR) is specified, the environment variable PGDATA is used.

Утилита может включать подсчет контрольных сумм на кластере.

10) Включите подсчет контрольных сумм. Параметр `-v` использовать не стоит, так как с ним выводится список всех файлов кластера, а их много.

```
postgres@tantor:~$ pg_checksums -e
Checksum operation completed
Files scanned: 948
Blocks scanned: 2817
Files written: 780
Blocks written: 2817
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster
```

11) Параметр `-c` проверяет блоки в существующих файлах данных на соответствие контрольным суммам, которые сохранены в их блоках.

Проверьте целостность файлов данных кластера:

```
postgres@tantor:~$ pg_checksums -c
Checksum operation completed
Files scanned: 948
Blocks scanned: 2817
Bad checksums: 0
Data checksum version: 1
```

Эта команда может использоваться для проверки нет ли повреждённых блоков.

Недостаток только тот, что экземпляр должен быть остановлен.

12) Запустите экземпляр кластера:

```
postgres@tantor:~$ pg_ctl start
waiting for server to start....
LOG: starting PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
LOG: listening on IPv4 address "127.0.0.1", port 5432
LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
LOG: database system was shut down at 13:25:56 MSK
LOG: database system is ready to accept connections
done
server started
```

Экземпляр использует порт 5432 для **Unix sockets** и на **локальном** сетевом интерфейсе.

13) Экземпляр также можно запускать командой `sudo systemctl start tantor-server-17`. И лучше использовать запуск с помощью `systemctl`. При запуске командой `pg_ctl start`, которую мы использовали, сообщения выводятся в `error output stream`, по умолчанию направленный на терминал пользователя операционной системы `postgres`.

Проверьте это:

```
postgres@tantor:~$ psql -c "\dconfig log_destination"
List of configuration parameters
Parameter      | Value
-----+-----
log_destination | stderr
(1 row)
```

При запуске с помощью `systemd` значение параметра то же (`log_destination=stderr`), но `error output stream` направлен на журнал операционной системы или процесс `syslog` (текстовый файл `/var/log/syslog`, куда собираются все сообщения от процессов операционной системы).

При промышленной эксплуатации в журнал могут передаваться большие объемы текста, и лучше использовать процесс сбора сообщений `logger` (включается параметром конфигурации `logging_collector=on`), который работает в асинхронном режиме и не приводит к задержкам в работе процессов. Конфигурирование журнала сообщений рассматривается в отдельной главе курса.

Часть 3. Режим одного пользователя

1) Посмотрим использование режима одного пользователя. Он используется в редких случаях.

Остановите экземпляр кластера:

```
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
```

Кроме сообщений об остановке **от утилиты** `pg_ctl` на экран будут выданы сообщения, которые обычно выводятся в диагностический журнал.

2) Запустите один процесс, который будет принимать наши команды в одной сессии:

```
postgres@tantor:~$ postgres --single
PostgreSQL stand-alone backend 17.5
```

3) Появится промпт. **Команды типа SELECT выдают результат не в обычном виде, а с диагностическими данными. Также команды необязательно завершать и посылать на выполнение символом ";"**.

Дайте команду `SELECT`:

```
backend> select tantor_version()
 1: tantor_version (typeid = 25, len = -1, typmod = -1, byval = f)
----
 1: tantor_version = "Tantor Special Edition 17.5.0" (typeid = 25, len = -1,
typmod = -1, byval = f)
----
```

4) Дайте команду `reindex system`:

```
backend> reindex system
```

5) Для выхода из сессии нужно набрать на клавиатуре комбинацию клавиш **<Ctrl+D>**.

Команды `psql` (начинающиеся на обратный слэш, например, команда выхода из `psql` `"\q"`) и их синонимы (`quit`, `exit` которые являются синонимами `\q`), не работают, так как мы работаем не в утилите `psql`.

Отсоединитесь от кластера, набрав комбинацию клавиш **<Ctrl+D>**:

```
backend> <Ctrl+D> LOG: checkpoint starting: shutdown immediate
LOG: checkpoint complete: wrote 145 buffers (0.9%); 0 WAL file(s) added, 0
removed, 1 recycled; write=0.007 s, sync=0.070 s, total=0.086 s; sync files=283,
longest=0.012 s, average=0.001 s; distance=5719 kB, estimate=5719 kB;
lsn=0/208C110, redo lsn=0/208C110
postgres@tantor:~$
```

Примечание: если вы по ошибке набрали **<Ctrl+z>** вместо **<Ctrl+D>** (EOF), то вы приостановили процесс и отправили его в фоновый режим. Вернуть процесс в режим `foreground` и получить возможность правильно завершить работу процесса можно командой `fg postgres`.

Пример:

```
postgres@tantor:~$ postgres --single

PostgreSQL stand-alone backend 17.5
backend> ^z
```

```
[1]+  Stopped                  postgres --single
postgres@tantor:~$ fg postgres
postgres --single
<ENTER>
backend> <Ctrl+D>
СООБЩЕНИЕ:  начата контрольная точка: shutdown immediate
```

Примечание: текст в сообщении "shutdown immediate" относится к свойствам контрольной точки, а не к режиму остановки экземпляра. При остановке экземпляра в режиме immediate (команда `pg_ctl stop -m immediate`) контрольная точка не выполняется.

Текст в сообщениях о контрольной точке (после `LOG: checkpoint starting:`) означает:

`shutdown`: контрольная точка вызвана остановкой экземпляра.

`immediate`: выполнить контрольную точку с максимальной скоростью, игнорируя значение параметра `checkpoint_completion_target`.

`force`: выполнить контрольную точку, даже если с прошлой контрольной точки в WAL ничего не было записано (в кластере не было активности), такое происходит если экземпляр останавливается (`shutdown`) или в конце восстановления (`end-of-recovery`).

`wait`: ждать завершения контрольной точки перед тем как вернуть управление процессу, вызвавшему контрольную точку (без `wait` процесс запустит контрольную точку и продолжит работать дальше).

`end-of-recovery`: контрольная точка по окончании наката журналов (восстановление по WAL).

`xlog`: контрольная точка вызвана достижением `max_wal_size` ("по размеру").

`time`: контрольная точка вызвана достижением `checkpoint_timeout` ("по времени").

6) Запустите экземпляр из-под пользователя root:

```
postgres@tantor:~$ su -
Password: root
root@tantor:~# systemctl start tantor-se-server-17
root@tantor:~#
```

7) Выйдите из терминала root (вместо `exit` можно набрать комбинацию клавиш

`<Ctrl+D>`):

```
root@tantor:~# exit
logout
postgres@tantor:~$
```

8) Остановите экземпляр. Независимо от того, как он запускался, его можно остановить утилитой `pg_ctl`:

```
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
```

Часть 4. Передача параметров экземпляру в командной строке

1) Посмотрим, как передавать параметры конфигурации для запуска экземпляра в командной строке. Установим параметр `work_mem` в значение 8 мегабайт. Часть параметров конфигурации можно задать только передав их в командной строке.

Запустите следующую команду:

```
postgres@tantor:~$ pg_ctl start -o "--work_mem=8MB" -l logfile.log
waiting for server to start....
[19479] LOG:  starting PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc
(Astra 12.2.0-14.astra3) 12.2.0, 64-bit
[19479] LOG:  listening on IPv6 address ":::1", port 5432
[19479] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
[19482] LOG:  database system was shut down at 13:59:08 MSK
[19479] LOG:  database system is ready to accept connections
```

2) Проверьте, что параметр установился:

```
postgres@tantor:~$ psql -c "show work_mem"
 work_mem
-----
      8MB
(1 row)
```

Часть 5. Локализация

1) После создания кластера проверим, удовлетворительно ли работает сортировка:

```
postgres=# SELECT n FROM unnest(ARRAY['a', 'e', 'ë', 'Ж', 'я', 'Ё', 'Е']) n ORDER
BY n;
```

```

n
---
a
e
E
ë
Ё
Ж
я
(7 rows)
```

```
postgres=# SELECT n FROM unnest(ARRAY['a', 'e', 'ë', 'Ж', 'я', 'Ё', 'Е']) n ORDER
BY n COLLATE "ru-x-icu";
```

```

n
---
a
e
E
ë
Ё
Ж
я
(7 rows)
```

2) Посмотрим, какие типы сортировок поддерживала операционная система при создании кластера:

```
postgres=# select collname from pg_collation where collname like '%ru%RU%';
 collname
```

```

-----
ru_RU
ru_RU.cp1251
ru_RU.iso88595
ru_RU.utf8
ru_RU
ru_RU
ru-RU-x-icu
(7 rows)
```

Часть 6. Однобайтные кодировки

Команды, приведенные ниже в этой части, не нужно выполнять, с ними можно

ознакомиться:

```
1) postgres=# select collname from pg_collation where collname like
'%ru%RU%';
 collname
-----
 ru_RU
 ru_RU.cp1251
 ru_RU.iso88595
 ru_RU.utf8
 ru_RU
 ru_RU
 ru-RU-x-icu
(7 rows)
```

2) Создание базы данных с другим типом сортировки:

```
postgres=# create database lab01iso88595 LC_COLLATE = 'ru_RU.iso88595';
ERROR:  encoding "UTF8" does not match locale "ru_RU.iso88595"
DETAIL:  The chosen LC_COLLATE setting requires encoding "ISO_8859_5".
```

Ошибка указывает на то, что сортировка связана с кодировкой.

3) Укажем кодировку:

```
postgres=# create database lab01iso88595 LC_COLLATE = 'ru_RU.iso88595'
ENCODING='ISO_8859_5';
ERROR:  encoding "ISO_8859_5" does not match locale "en_US.UTF-8"
DETAIL:  The chosen LC_CTYPE setting requires encoding "UTF8".
```

Ошибка указывает на то, что `ctype` также связан с кодировкой.

4) Попробуем ещё:

```
postgres=# create database lab01iso88595 LC_COLLATE = 'ru_RU.iso88595'
LC_CTYPE='ru_RU.iso88595';
ERROR:  encoding "UTF8" does not match locale "ru_RU.iso88595"
DETAIL:  The chosen LC_CTYPE setting requires encoding "ISO_8859_5".
```

Убеждаемся, что выбранный `ctype` требует задания кодировки для создаваемой базы данных.

5) Укажем все три параметра:

```
postgres=# create database lab01iso88595 LC_COLLATE = 'ru_RU.iso88595'
LC_CTYPE='ru_RU.iso88595' ENCODING='ISO_8859_5';
ERROR:  new encoding (ISO_8859_5) is incompatible with the encoding of the
template database (UTF8)
HINT:  Use the same encoding as in the template database, or use template0 as
template.
```

Ошибка указывает на то, что база данных `template1` не может использоваться, единственный шаблон, который может использоваться это `template0`.

6) Укажем имя шаблона:

```
postgres=# create database lab01iso88595 LC_COLLATE = 'ru_RU.iso88595'
LC_CTYPE='ru_RU.iso88595' ENCODING='ISO_8859_5' TEMPLATE= template0;
CREATE DATABASE
```

При создании базы данных с кодировкой, отличной от кодировки по умолчанию, для кластера пришлось указать все четыре параметра.

7) Подключимся к новой базе данных и проверим, правильно ли работает сортировка с однобайтной кодировкой. Зададим её явно, но можно было не указывать, так как для этой базы данных, это значение сортировки используется по умолчанию:

```
postgres=# \c lab01iso88595
You are now connected to database "lab01iso88595" as user "postgres".
lab01iso88595=# SELECT n FROM unnest(ARRAY['a', 'e', 'ë', 'ж', 'я', 'Ё', 'Е']) n
ORDER BY n COLLATE "ru_RU.iso88595";
 n
---
 a
 e
 Е
 ë
 Ё
 ж
 я
(7 rows)
```

Работает правильно, так же как и с кодировкой UTF-8.

Часть 7. Использование утилит управления

Познакомимся с утилитами командной строки, которые являются оболочками команд SQL. Возможно, их будет удобно использовать.

1) Посмотрите параметры утилиты создания баз данных. У утилит командной строки Линукс принято иметь параметр (ключ) с названием `--help` или `-h` с кратким описанием параметров.

```
postgres@tantor:~$ createdb --help
```

Создайте базу данных с именем `lab01database`:

```
postgres@tantor:~$ createdb lab01database
```

Ошибки не выдано, значит база данных создана.

2) Посмотрите список баз данных кластера и их табличных пространств по умолчанию утилитой `oid2name`. Проверьте, что база данных `lab01database` есть в списке:

```
postgres@tantor:~$ oid2name
All databases:
  Oid  Database Name  Tablespace
-----
 16798 lab01database  pg_default
 16797 lab01iso88595  pg_default
    5  postgres      pg_default
    4  template0     pg_default
    1  template1     pg_default
```

3) Создайте пользователя с именем `lab01user`, с таким же паролем и с атрибутами, позволяющими подсоединяться к базам данных кластера, и атрибутом суперпользователя:

```
postgres@tantor:~$ createuser lab01user --login --superuser -P
Enter password for new role: lab01user
Enter it again: lab01user
postgres@tantor:~$
```

4) Запустите утилиту выгрузки данных из кластера и в режиме выгрузки глобальных объектов: Глобальные объекты - это общие объекты для всех баз данных кластера. По умолчанию утилита выводит создаваемые команды в `stdout` (на экран терминала).

```
postgres@tantor:~$ pg_dumpall -g
--
-- PostgreSQL database cluster dump
--
SET default_transaction_read_only = off;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
--
-- Roles
--
CREATE ROLE lab01user;
ALTER ROLE lab01user WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
NOREPLICATION NOBYPASSRLS PASSWORD 'SCRAM-SHA-256$4096:...';
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION
BYPASSRLS;
```

В выданных командах будет присутствовать команда **воссоздания пользователя**, которого только что создали.

5) Выполните вакуумирование всех баз данных и заморозку строк:

```
postgres@tantor:~$ vacuumdb -a -F
vacuumdb: vacuuming database "lab01database"
vacuumdb: vacuuming database "lab01iso88595"
vacuumdb: vacuuming database "postgres"
vacuumdb: vacuuming database "template1"
```

6) Проверьте, что кластер запущен и принимает соединения

```
postgres@tantor:~$ pg_isready
/var/run/postgresql:5432 - accepting connections
```

Часть 8. Настройка терминального клиента psql

1) Проверьте, что вы в терминале пользователя postgres, посмотрев промпт терминала командной строки:

```
postgres@tantor:~$
```

2) Запустите psql и выйдите из интерактивного режима утилиты. Для выхода можно использовать команду `\q`, либо комбинацию клавиш `<Ctrl+D>`, либо `quit`, либо `exit`.

```
postgres@tantor:~$ psql
psql (17.5)
Type "help" for help.
postgres=# \q
postgres@tantor:~$
```

Обратите внимание на приглашения (prompt) `psql` и терминала, они отличаются. Это пригодится, чтобы не вводить SQL-команды в терминале операционной системы и наоборот.

3) Посмотрите содержимое файла `.psqlrc`:

```
postgres@tantor:~$ cat ~/.psqlrc
\setenv PAGER 'less -XS'
\setenv PSQL_EDITOR '/usr/bin/mcedit'
\set ON_ERROR_ROLLBACK interactive
\pset pager off
postgres@tantor:~$
```

При редактировании процедур, функций, представлений в psql будет использоваться редактор `mcedit`. Для постраничного вывода будет использоваться утилита `less`. Однако, **постраничный вывод отключён**, так как он не удобен.

Для добавления строк в конец файла можно использовать команду:

```
echo "\setenv PAGER 'less -XS'" >> ~/.psqlrc
```

4) Возможно использование и графических редакторов. В AstraLinux стандартно установлен графический редактор `kate`. Однако, если вы используете утилиту `su` для переключения терминала в другого пользователя операционной системы, графический редактор не запустится. В этом случае можно вместо `su` использовать команды, приведенные ниже. Команды в этом пункте приведены для справки и их не нужно выполнять.

```
postgres@tantor:~$ exit
logout
root@tantor:/home/astra# exit
exit
astra@tantor:~$ ssh -X postgres@localhost
The authenticity of host 'localhost (:::1)' can't be established.
ECDSA key fingerprint is SHA256:12VsUcC5hw5I1zr015AJ8C+xsN0m5h+I1U2M/xdNg6o.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
postgres@localhost's password: postgres
/usr/bin/xauth: file /var/lib/postgresql/.Xauthority does not exist
postgres@tantor:~$ export PSQL_EDITOR=kate
postgres@tantor:~$ pg_ctl stop
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
Убрать из ~/.psqlrc строку \setenv PSQL_EDITOR /usr/bin/mcedit
```

5) Запустите `psql` и **включите постраничный вывод**:

```
postgres@tantor:~$ psql
psql (17.5)
Type "help" for help.
postgres=# \pset pager on
```

Подключившись по ssh не стоит запускать экземпляр утилитой `pg_ctl start`, так как после закрытия соединения ssh экземпляр остановится. Причина - родительский процесс, запустивший процесс `postgres` останавливается. При подсоединении по ssh запускать экземпляр стоит командой `sudo systemctl start tantor-se-server-17`.

6) Посмотрите подсказку по командам `psql`, набрав команду `\?`, и промотайте нажатием клавиши `<Enter>` на клавиатуре до подраздела Query Buffer:

```
postgres=# \?
...
Query Buffer
\e [FILE] [LINE]      edit the query buffer (or file) with external editor
\ef [FUNCNAME [LINE]] edit function definition with external editor
\ev [VIEWNAME [LINE]] edit view definition with external editor
\p                   show the contents of the query buffer
\r                   reset (clear) the query buffer
\s [FILE]            display history or save it to file
\w FILE              write query buffer to file
```

Можно пользоваться клавишами `z` - промотать экран вверх, `b` - промотать экран вниз, `q` - выйти.

Также можно проматывать буфер терминала клавишами `<Shift+PgUp>` `<Shift+PgDn>`.

7) Если удобнее читать подсказку на русском языке, установите переменную окружения `LC_MESSAGES`, которой устанавливается язык сообщений утилит. Это можно сделать на уровне терминала, настройка и будет действовать до тех пор пока не закроете терминал.

Нажмите на клавиатуре комбинацию клавиш `<Ctrl+D>` (или наберите команду `\q` и клавишу `<Enter>`). Удобно использовать `<Ctrl+D>`, так как она универсальна и быстрее набирается.

Наберите команду:

```
postgres@tantor:~$ export LC_MESSAGES=ru_RU.utf8
unset LANGUAGE
```

8) Если вы хотите чтобы настройка действовала постоянно, то наберите команды:

```
postgres@tantor:~$ cp .bash_profile .bash_profile.OLD
echo "export LC_MESSAGES=ru_RU.utf8" >> ~/.bash_profile
echo "unset LANGUAGE" >> ~/.bash_profile
```

9) Если вместо символов `">>"` набрать `">"`, то содержимое файла затрётся. Двойной символ добавляет строку в конец файла. В домашнем каталоге может присутствовать файл `.profile`. Этот файл неудобен тем, что если в домашнем каталоге присутствует файл `.bash_profile` или `.bash_login`, то файл `.profile` не действует.

Запустите `psql` и повторите команду `\?`. История команд сохраняется и можно, набирая на клавиатуре стрелку вверх или вниз, выбирать команды из истории, а нажимая `<Enter>` - их повторять.

```

postgres@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
postgres=# \?
Буфер запроса

\e [ФАЙЛ] [СТРОКА] править буфер запроса (или файл) во внешнем редакторе
\ef [ФУНКЦИЯ [СТРОКА]] править определение функции во внешнем редакторе
\ev [VIEWNAME [LINE]] править определение представления во внешнем редакторе
\p вывести содержимое буфера запросов
\r очистить буфер запроса
\s [ФАЙЛ] вывести историю или сохранить её в файл
\w ФАЙЛ записать буфер запроса в файл

:q
    
```

Если вы хотите прервать отображение подсказки, нажмите клавишу "q"

10) Прочтите выделенный текст. Команды `\p` `\r` обычно забывают или не знают о них, но они полезны.

Как `psql` взаимодействует с программой редактора? Когда вы набираете команды `\e` `\ef` `\ev` запускается редактор и ему `psql` передает текст того, что хотите редактировать, и путь к временному файлу, который вы обычно не видите. В примере ниже название файла высвечивается на первой строке картинки как `/tmp/psql/edit.6652.sql`

Дальше средствами редактора вы редактируете текст и нажимаете в редакторе "сохранить" отредактированное и "закрыть" редактор. Редактор сохраняет текст в файл и `psql` получает уведомление, что редактор закрыт. Скрыто от вас `psql` открывает файл и загружает его в буфер, точно так же, как будто вы набрали содержимое файла на клавиатуре.

Нюанс: если в конце команды, когда вы находились в редакторе, вы не поставили точку с запятой и переход на новую строку в конце набранной или редактируемой команды, или уже находясь в `psql` вы её не набрали, то команда не пойдёт на выполнение и вы будете продолжать заполнять буфер. Этот нюанс может затруднять использование команд `\e` `\ef` и `\ev`, склоняя к использованию графических средств типа `pgAdmin`.

11) Вызовите редактор создания представления командой `\ev` наберите команду, как показано ниже, нажмите клавишу F2 (сохранить) F10 (выйти). При желании, вы можете выбрать редактор, который вам удобнее. В редакторе `kate`, который возможно использовать в AstraLinux, быстрые комбинации клавиш: `<Ctrl+S>` - сохранить, `<Ctrl+Q>` - выйти из редактора.

```

/tmp/psql/edit.6652.sql [-M--] 0 L:[ 1+ 2
CREATE OR REPLACE VIEW public.lab01view AS
SELECT now();
    
```

```

postgres=# \ev
CREATE VIEW
    
```

12) Командой `\p` посмотрите последнюю команду. Команда была получена `psql` из редактора.

После посылки команд на выполнение:

```

postgres=# \p
    
```

```
CREATE VIEW lab01view AS
SELECT now();
```

13) Также можно посмотреть определение представление или подпрограммы (routines к которым относятся процедуры и функции):

```
\sf[+] ИМЯ_ФУНКЦИИ показать определение функции
\sv[+] ИМЯ_ПРЕДСТ показать определение представления
```

Наберите:

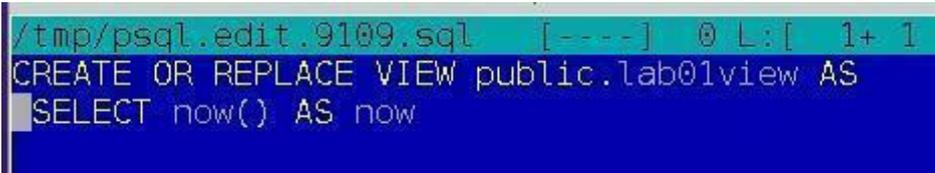
```
\sv 1<TAB><ENTER>
```

Где <TAB> - клавиша табуляции, <ENTER> - тоже клавиша на клавиатуре.

После нажатия клавиши <TAB> psql дополнит название представления. Если представлений, начинающихся на букву "l" много (или вообще нет таких), то не дополнит. В этом случае второе нажатие клавиши <TAB> высветит список кандидатов. Вы сможете набрать еще несколько символов и снова нажать <TAB>, и потом послать на выполнение то, что набрали нажав клавишу <ENTER>.

```
postgres=# \sv lab01view
CREATE OR REPLACE VIEW public.lab01view AS
SELECT now() AS now
postgres=#
```

Обратите внимание, что точки с запятой в конце команды нет. При открытии редактора она также не появится. После "now" надо вставить ; и возврат каретки.



Мы рассмотрели детали работы самого неочевидного функционала psql - взаимодействия с редактором. Остальная информация гораздо проще.

Часть 9. Использование терминального клиента psql

1) Выполните команды:

```
postgres=# BEGIN TRANSACTION;
BEGIN
```

2) Мы начали транзакцию. Обратите внимание, что промпт изменился - появился символ звездочки. В psql с промптом по умолчанию вы видите, если ли открытая транзакция, чтобы принять решение, зафиксировать ли ее.

Дальше наберем команду в несколько строк.

Наберите SELECT:

```
postgres=# select
```

3) Обратите внимание на то, что промпт опять изменился: вместо символа равенство появилось тире. Донаберите команду и завершите команду точкой с запятой:

```
postgres-# tantor_version();
tantor_version
-----
Tantor Special Edition 17.5.0
(1 row)
```

4) Обратите внимание - промпт опять изменился: вместо символа тире вернулся символ равенства. Это означает, что в буфере нет незавершенной команды, и вы будете набирать первую строку команды.

Наберите ошибочную команду и пошлите на выполнение точкой с запятой:

```
postgres=# ffff;
ERROR:  syntax error at or near "ffff"
LINE 1: ffff;
```

Выдалась ошибка синтаксиса. Обратите внимание на то, что вместо звёздочки, означающей открытую транзакцию, появился символ восклицательного знака. Это означает, что транзакция ещё открыта, но она перешла в состояние сбоя, а в таком состоянии транзакция не может зафиксироваться, а только откатиться. В состоянии сбоя транзакции переходят нечасто, а только после некоторых ошибок, которые считаются серьезными настолько, что зафиксировать операторы, накопившиеся в транзакции, нельзя. Например, ошибки сериализации доступа. Что опасного в команде "ffff"? Её получает серверный процесс и видит, что это что-то совсем дикое, не может же программист написать такую команду. Серверный процесс рассчитывает, что ему даёт команды приложение, написанное программистом и оттестированное. Поэтому считает, что нужно перевести транзакцию в состояние сбоя.

5) Проверим, что если послать на выполнение правильную команду. Наберите:

```
postgres=!# select 1;
ERROR:  current transaction is aborted, commands ignored until end of transaction
block
```

Выдается ошибка, что команда не выполнена, и любые команды будут игнорироваться серверным процессом, пока клиент "добровольно" не завершит транзакцию.

6) Завершаем транзакцию одной из двух команд завершения транзакций:

```
postgres=# COMMIT;
ROLLBACK
```

Обратите внимание, что транзакция переведенная в состояние сбоя, не может зафиксироваться, только откатиться полностью или до точки сохранения, если она была установлена. На команду `COMMIT` серверный процесс возвращает: "транзакция завершена путем отката".

Есть параметр `ON_ERROR_ROLLBACK`, он позволяет не терять результаты выполненных команд. Этот параметр заставляет `psql` устанавливать после каждой команды точку сохранения (SAVEPOINT), что нежелательно, так как увеличивает использование счетчика транзакций (xid). Если его устанавливать, то лучше в значение `INTERACTIVE`, тогда точки сохранения будут устанавливаться, если работать в `psql` интерактивно.

7) Установите этот параметр:

```
postgres=# \set ON_ERROR_ROLLBACK INTERACTIVE
```

8) Повторите команды из предыдущего примера:

```
postgres=# BEGIN;
BEGIN
postgres=# select 1;
?column?
-----
      1
(1 строка)
postgres=# ffff;
ERROR:  syntax error at or near "ffff"
СТРОКА 1: ffff;
          ^
postgres=# COMMIT;
COMMIT
```

Транзакция была закрыта фиксацией, а не откатом.

9) Посмотрим, как `psql` обрабатывает свои команды - что он посылает серверному процессу, чтобы вывести красивый результат. Посмотрим, какие роли есть в кластере. На английском языке это бы звучало "describe user", сокращения по первым буквам слов "du". Добавим обратный слэш - общее начало всех команд утилиты `psql`. Если без обратного слэша, это команда SQL и посылается на выполнение серверному процессу как текст. Для посылки на выполнение используется точка с запятой ";" - иначе как `psql` узнать, что вы завершили набирать команду.

Наберите:

```
postgres=# \du
```

Имя роли	Список ролей	Атрибуты
lab01user	Суперпользователь,	Создаёт роли, Создаёт БД
postgres	Суперпользователь,	Создаёт роли, Создаёт БД, Репликация, Пропускать RLS

10) Установите параметр `psql`, который покажет нам какую команду `psql` сам формирует и посылает на выполнение:

```
postgres=# \set ECHO_HIDDEN on
```

11) Повторите команду:

```
postgres=# \du
***** ЗАПРОС *****
SELECT r.rolname, r.rolsuper, r.rolinherit,
       r.rolcreatorole, r.rolcreatedb, r.rolcanlogin,
       r.rolconlimit, r.rolvaliduntil
, r.rolreplication
, r.rolbypassrls
FROM pg_catalog.pg_roles r
WHERE r.rolname !~ '^pg_'
ORDER BY 1;
*****
```

Список ролей

Имя роли	Атрибуты
lab01user	Суперпользователь, Создаёт роли, Создаёт БД
postgres	Суперпользователь, Создаёт роли, Создаёт БД, Репликация, Пропускать RLS

12) Скопируйте и вставьте (copy-paste) текст команды. Для этого можно использовать комбинации клавиш `<Ctrl+Shift+c>` `<Ctrl+Shift+v>`

```
postgres=# SELECT r.rolname, r.rolsuper, r.rolinherit, r.rolcreatorole,
r.rolcreatedb, r.rolcanlogin, r.rolconlimit, r.rolvaliduntil, r.rolreplication,
r.rolbypassrls
FROM pg_catalog.pg_roles r
WHERE r.rolname !~ '^pg_'
ORDER BY 1;
```

rolname	rolsuper	rolinherit	rolcreatorole	rolcreatedb	rolcanlogin	rolconlimit	rolvaliduntil	rolreplication	rolbypassrls
lab01user	t	t	t	t	t	-1		f	f
postgres	t	t	t	t	t	-1		t	t

(2 строки)

Видно, какую информацию получает `psql` и можно сравнить с тем, как он интеллектуально ее отображает: `psql` не отобразил атрибуты `INHERIT` и `LOGIN`. Почему? Потому что это значения по умолчанию при создании роли. Значения по умолчанию не отображаются. Будут отображаться их обратные значения: "Не наследуется, Вход запрещён". Эта особенность интуитивна непонятна, поэтому мы остановились на ней подробно.

13) Посмотрите командой `\?` справку по команде `\connect` (сокращённая версия команды `\c`)

Соединение:

```
\c[onnect] {[БД|- ПЛЬЗОВАТЕЛЬ|- СЕРВЕР|- ПОРТ|-] | conninfo}
                подключиться к другой базе данных
                (текущая: "postgres")
\conninfo
                информация о текущем соединении
```

14) Попробуйте различные комбинации подсоединения. Клавиша табуляции позволяет заканчивать параметр, так как `psql` в текущем подсоединении имеет доступ к списку имён баз данных и пользователей. Цель этой последовательности соединений - запомнить порядок следования параметров команды `\c: база пользователь хост порт`. Если какой-то параметр хотите оставить прежним, то замените его на тире. `<TAB><ENTER>` - клавиши табуляции и возврат каретки (новая строка) на клавиатуре.

```
postgres=# \c la<TAB><ENTER>
Вы подключены к базе данных "lab01database" как пользователь "postgres".
lab01database=# \c - la<TAB><ENTER>
Вы подключены к базе данных "lab01database" как пользователь "lab01user".
lab01database=# \c - - localhost
Сейчас вы подключены к базе данных "lab01database", как пользователь "lab01user"
(сервер "localhost": адрес "127.0.0.1", порт "5432").
lab01database=# \c - - - 5432
Вы подключены к базе данных "lab01database" как пользователь "lab01user".
lab01database=# \c postgres p<TAB><ENTER>
Вы подключены к базе данных "postgres" как пользователь "postgres".
```

15) Посмотрим, как получать результат выборки в формате веб-страницы и просматривать ее в браузере. Откройте **НОВОЕ** окно терминала (пользователь операционной системы **astra**).

16) Запустите `psql`:

```
astra@tantor:~$ psql
psql (17.5)
Type "help" for help.
```

17) Установите формат вывода HTML:

```
postgres=# \pset format html
Output format is html.
```

18) Перенаправьте вывод в файл с названием `file.html`:

```
postgres=# \o file.html
```

19) Дайте любую команду, результат которой неудобно читать в терминале:

```
postgres=# show all;
```

20) Отключите вывод в файл:

```
postgres=# \o
```

21) Запустите, на выходя из `psql`, окно браузера:

```
postgres=# \! xdg-open file.html
```

22) Подождите, пока не запустится окно браузера. Закройте `psql`:

```
postgres=# \q
```

23) Закройте окно терминала:

```
postgres@tantor:~$ <CTRL+d>
```

name	setting	description
allow_in_place_tablespaces	off	Allows tablespaces directly inside pg_tblspc, for testing.
allow_system_table_mods	off	Allows modifications of the structure of system tables.
application_name	psql	Sets the application name to be reported in statistics and logs.
archive_cleanup_command		Sets the shell command that will be executed at every restart point.
archive_command	(disabled)	Sets the shell command that will be called to archive a WAL file.
archive_mode	off	Allows archiving of WAL files using archive_command.
archive_timeout	0	Forces a switch to the next WAL file if a new file has not been started within N seconds.
array_nulls	on	Enable input of NULL elements in arrays.
authentication_timeout	1min	Sets the maximum allowed time to complete client authentication.
auto_explain.log_analyze	on	Use EXPLAIN ANALYZE for plan logging.
auto_explain.log_buffers	on	Log buffers usage.
auto_explain.log_format	text	EXPLAIN format to be used for plan logging.
auto_explain.log_level	log	Log level for the plan.
auto_explain.log_min_duration	5s	Sets the minimum execution time above which plans will be logged.
auto_explain.log_nested_statements	off	Log nested statements.
auto_explain.log_settings	off	Log modified configuration parameters affecting query planning.
auto_explain.log_timing	on	Collect timing data, not just row counts.
auto_explain.log_triggers	off	Include trigger statistics in plans.
auto_explain.log_verbose	on	Use EXPLAIN VERBOSE for plan logging.
auto_explain.log_wal	off	Log WAL usage.

24) Закройте окно браузера и вернитесь в окно `psql`. Посмотрим, какие ещё есть форматы вывода. Наберите в `psql`:

```
postgres=# \pset format aaa
\pset: allowed formats are aligned, asciidoc, csv, html, latex, latex-longtable,
troff-ms, unaligned, wrapped
```

25) Выберите формат `aligned`, он используется по умолчанию:

```
postgres=# \pset format aligned
Output format is aligned.
```

26) Выполните команду:

```
postgres=# SHOW ALL;
```

В конце экрана высветится двоеточие. Нажмите на клавиатуре клавиши `z z b q` и посмотрите эффект.

`z` - следующая страница, `b` - предыдущая, `q` - завершить вывод и вернуть промпт.

27) Выполните команду:

```
postgres=# \pset format wrapped
Output format is wrapped.
```

28) Выполните команду:

```
postgres=# SHOW ALL;
```

Нажмите на клавиатуре клавиши `z z b h`. Прочтите описание доступных клавиш. Закрепите навыки проматывания результата.

29) Сравните отличия. Возможно, формат `wrapped` (перенос по словам) будет удобнее `aligned`.

30) Проверим, как можно, не выходя из `psql`, выполнять команды операционной системы. Команда Линукс `"pwd"` показывает текущую директорию.

Выполните команду `"pwd"` или `"ls"` (выдаёт список файлов) без выхода из `psql`:

```
postgres=# \! pwd
```

/var/lib/postgresql

31) Установите цветной промпт, который будет высвечивать номер (pid) серверного процесса серым цветом:

```
\set PROMPT1 '%[%033[0;90m%][%p] [%033[0m%]
[%033[0;31m%] %n [%033[0m%] @ [%033[0;34m%] %m [%033[0m%] : [%033[0;32m%] %> [%033[0m%]
[%033[0;36m%] %~ [%033[0m%] _ [%033[0;33m%] [%033[5m%] %x [%033[0m%] [%033[0m%] %R%# '
\set PROMPT2 '%[%033[0;90m%][%1] [%033[0m%]
[%033[0;31m%] %n [%033[0m%] @ [%033[0;34m%] %m [%033[0m%] : [%033[0;32m%] %> [%033[0m%]
[%033[0;36m%] %~ [%033[0m%] _ [%033[0;33m%] [%033[5m%] %x [%033[0m%] [%033[0m%] %R%# '
```

Статус транзакции отображается мигающими символами * и ! привлекая внимание.

```
[22358] postgres@[local]:5432 ~ =#
[22358] postgres@[local]:5432 ~ =# BEGIN;
BEGIN
[22358] postgres@[local]:5432 ~ *=# err;
ERROR: syntax error at or near "err"
LINE 1: err;
      ^
[22358] postgres@[local]:5432 ~ !=# COMMIT;
ROLLBACK
[22358] postgres@[local]:5432 ~ =# \! ps -ef | grep 22358
postgres 22358  501  0 20:46 ?        00:00:00 postgres: postgres postgres [local] idle
postgres 24883 22357  0 20:53 pts/0    00:00:00 sh -c ps -ef | grep 22358
postgres 24885 24883  0 20:53 pts/0    00:00:00 grep 22358
[22358] postgres@[local]:5432 ~ =#
```

Справка, что означают символы, если захотите создать свой промпт:

- %p** номер серверного процесса
- %n** роль. (может изменяться во время сессии командой `SET SESSION AUTHORIZATION;`)
- %m** имя хоста или `[local]`, если соединение осуществляется через Unix-сокеты
- %>** номер порта экземпляра
- %/** имя базы данных
- %~** имя базы данных. Если это база данных по умолчанию, то вместо имени отображается `~`
- %#** для суперпользователя - символ `#`, для остальных ролей - символ `>%!` номер строки в буфере ввода.
- %R** для `PROMPT1` отображает `=`, если сессия находится в неактивной ветви условного блока `@` в режиме однострочного ввода `^`, если сессия отключена от базы данных - `!`
- для `PROMPT2` если команда не завершена -
- если есть незакрытый комментарий `*` если есть незакрытая кавычка, то `'`
- если есть незавершенная двойная кавычка, то `"`
- если есть начатая, но незавершенная строка `$` (обычно при наборе текста функций), то `$`
- если есть левая скобка, а правая скобка не введена, то `(`

Символы, которые отображает `PROMPT2` важны тем, что забыв набрать закрывающий апостроф, сколько `<ENTER>`; или `\r` не набирайте, никакой реакции не будет, пока не наберёте апостроф:

```
postgres@postgres=#
postgres@postgres=# SELECT 1 from pg_se
pg_seclabel pg_seclabels pg_sequence pg_sequences pg_settings
postgres@postgres=# SELECT 1 from pg_settings WHERE name = 'ЗабывлиЗакретьАпострофом ;
postgres@postgres:# что ни набирай
postgres@postgres:# \r
postgres@postgres:# ;
postgres@postgres:# ничего не поможет пока не наберете апостроф '
postgres@postgres=# ;
?column?
-----
(0 rows)
postgres@postgres=#
```

Если нужно отобразить роль и базу:

```
\set PROMPT1 '%[%033[0;31m%] %n%[%033[0m%]@%[%033[0;36m%] %/%[%033[0m%]
%[%033[0;33m%] %[%033[5m%] %x%[%033[0m%] %[%033[0m%] %R%# '
\set PROMPT2 '%[%033[0;31m%] %n%[%033[0m%]@%[%033[0;36m%] %/%[%033[0m%]
%[%033[0;33m%] %[%033[5m%] %x%[%033[0m%] %[%033[0m%] %R%# '
postgres@postgres=#
```

32) Посмотрите, как выводится результат запроса:

```
postgres=# select * from pg_user;
username | usesysid | usecreatedb | usesuper | userepl |
-----+-----+-----+-----+-----+
postgres |         10 | t           | t        | t        |
(1 строка)
```

33) Установите стиль отрисовки линий символами unicode:

```
postgres=# \pset linestyle unicode
```

Установлен стиль линий: unicode.

Повторим запрос (нажмите два раза на клавиатуре стрелку вверх и клавишу <ENTER>)

```
postgres=# select * from pg_user;
username | usesysid | usecreatedb | usesuper | userepl |
-----+-----+-----+-----+
postgres |         10 | t           | t        | t        |
(1 строка)
```

34) Поменяйте стиль отображения границ:

```
postgres=# \pset border 0
```

Стиль границ: 0.

35) Повторите запрос:

```
postgres=# select * from pg_user;
username  usesysid  usecreatedb  usesuper  userepl
-----
postgres  10 t          t          t
```

Отображение стало более компактным.

36) Поменяйте стиль отображения границ:

```
postgres=# \pset border 2
```

Стиль границ: 2.

```
postgres=# select * from pg_user;
username | usesysid | usecreatedb | usesuper | userepl | usebypassrls | passwd | valuntil | useconfig |
-----+-----+-----+-----+-----+-----+-----+-----+-----+
postgres |         10 | t           | t        | t        | t            | ***** |          |            |
```

(3 строки)

Вы можете выбрать наиболее удобный для себя стиль вывода результатов выборки. Чтобы сделать его постоянным, можно отредактировать файл `~/.psqlrc`, добавив в этот файл команды, которые мы рассмотрели.

Часть 10. Восстановление сохраненного кластера

В пункте 4 части 2 мы сохранили прежний кластер перед созданием нового кластера.

Вернём кластер на место.

1) Остановите экземпляр:

```
postgres@tantor:~$ pg_ctl stop
```

2) Выполните команды:

```
postgres@tantor:~$ mkdir $PGDATA/../data.afterLAB1
postgres@tantor:~$ mv $PGDATA/* $PGDATA/../data.afterLAB1
postgres@tantor:~$ mv $PGDATA/../data.SAVE/* $PGDATA
```

3) Запустите экземпляр:

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
```

4) Проверьте работоспособность экземпляра:

```
postgres@tantor:~$ psql -c "select datname from pg_database;"
datname
-----
postgres
template1
template0
(3 строки)
```

Глава 2а. Архитектура

Часть 1. Транзакция в psql

1) Откроем терминал Fly на рабочем столе:

```
astra@tantor:~$ psql
psql (17.5)
Type "help" for help.
```

2) Введем "help", чтобы получить справку:

```
postgres=# help
You are using psql, the command-line interface to PostgreSQL.
Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help with psql commands
      \g or terminate with semicolon to execute query
      \q to quit
```

3) Создадим произвольную таблицу:

```
postgres=# CREATE TABLE a(id integer);
CREATE TABLE
```

4) Посмотрим, что получилось:

```
postgres=# \dt a
          Список отношений
Схема | Имя | Тип | Владелец
-----+-----+-----+-----
public | a   | таблица | postgres
(1 строка)
```

5) Откроем транзакцию:

```
postgres=# begin;
BEGIN
```

6) Вставим первую строчку. Обратите внимание, что с помощью табуляции можно дописывать ключевые слова и даже сложные конструкции.

```
postgres=# INSERT INTO a VALUES (1);
INSERT 0 1
```

Обратите внимание на появление звездочки в строке - это означает, что идет транзакция.

7) Попробуем во втором терминале увидеть первую строчку таблицы. Откроем второй терминал:

8) Запустите psql.

```
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
```

```
postgres=#
```

9) Обратимся к таблице:

```
postgres=# SELECT * FROM a;
 id
----
```

(0 строк)

Убедились - пока мы не видим первой строчки. Видны только зафиксированные данные. Грязное чтение не допускается.

10) В первом терминале зафиксируем транзакцию.

```
postgres=# COMMIT;
COMMIT
```

11) Во втором терминале обратимся к таблице еще раз.

```
postgres=# SELECT * FROM a;
 id
----
  1
(1 строка)
```

Теперь изменения таблицы зафиксированы.

Вывод - видны только те изменения, которые успешно зафиксированы.

Часть 2. Список фоновых процессов

1) Посмотрим, где находится директория PGDATA, где находятся файлы кластера БД.

```
postgres=# SHOW data_directory;
 data_directory
-----
 /var/lib/postgresql/tantor-se-17/data
(1 строка)
```

2) Выйдите в первом терминале из psql.

```
postgres=# \q
```

3) Чтобы посмотреть список процессов воспользуемся утилитой ps :

```
astra@tantor:~$
sudo -u postgres cat /var/lib/postgresql/tantor-se-17/data/postmaster.pid
466
/var/lib/postgresql/tantor-se-17/data
1713847705
5432
/var/run/postgresql
*
 1048641          0
ready
```

4) Возьмем PID = 466

```
astra@tantor:~$ sudo ps -o command --ppid 466
COMMAND
postgres: checkpointer
postgres: background writer
postgres: walwriter
postgres: autovacuum launcher
postgres: logical replication launcher
postgres: postgres postgres [local] idle
```

Цветом показаны системные фоновые процессы, остальные - серверные.

Список процессов можно увидеть также через представление pg_stat_activity.

5) Сделайте во втором терминале:

```
postgres=# SELECT pid, backend_type, backend_start FROM pg_stat_activity;
```

pid	backend_type	backend_start
527	autovacuum launcher	2035-07-25 07:48:25.435889+03
528	logical replication launcher	2035-07-25 07:48:25.441432+03
540	client backend	2035-07-25 07:48:51.242631+03
520	background writer	2035-07-25 07:48:25.403365+03
519	checkpointer	2035-07-25 07:48:25.402941+03
526	walwriter	2035-07-25 07:48:25.425135+03

(8 строк)

Часть 3. Буферный кэш, команда EXPLAIN

1) Во втором терминале добавим строки в таблицу "a":

```
postgres=# INSERT INTO a SELECT id FROM generate_series(1,10000) AS id;
INSERT 0 10000
```

2) В первом терминале перезагрузите сервер:

```
astra@tantor:~$ sudo systemctl restart tantor-se-server-17
```

3) Во втором терминале сделайте реконнект:

```
postgres=# \c
```

Вы подключены к базе данных "postgres" как пользователь "postgres".

4) С помощью команды EXPLAIN посмотрите, откуда берется информация:

```
postgres=# EXPLAIN (analyze, buffers) SELECT * FROM a;
               QUERY PLAN
-----
Seq Scan on a (cost=0.00..145.00 rows=10000 width=4) (actual time=0.035..1.952
rows=10000 loops=1)
  Buffers: shared read=45
Planning:
  Buffers: shared hit=16 read=6 dirtied=3
Planning Time: 0.428 ms
Execution Time: 2.948 ms
(6 строк)
```

Обратите внимание на строку **Buffers**. Информация была взята с диска или из страничного кэша операционной системы.

5) Сделайте эксперимент еще раз:

```
postgres=# EXPLAIN (analyze, buffers) SELECT * FROM a;
               QUERY PLAN
-----
Seq Scan on a (cost=0.00..145.00 rows=10000 width=4) (actual time=0.016..1.383
rows=10000 loops=1)
  Buffers: shared hit=45
Planning Time: 0.063 ms
Execution Time: 2.355 ms
(4 строки)
```

Информация изменилась. Теперь информация найдена в буферном кэше.

Часть 4. Журнал предзаписи. Где хранится?

В первом терминале выполните команду:

```
astra@tantor:~$ sudo ls -l /var/lib/postgresql/tantor-se-17/data/pg_wal
итого 360452
```

```
-rw----- 1 postgres postgres 16777216 Jun 26 14:00 00000001000000000000000002
-rw----- 1 postgres postgres 16777216 Jun 26 12:10 00000001000000000000000003
drwx----- 2 postgres postgres      4096 Jun 26 10:54 archive_status
drwx----- 2 postgres postgres      4096 Jun 26 10:54 summaries
```

Файлы журнала предзаписи находятся в директории `pg_wal` сегментами по 16 мегабайт.

Часть 5. Контрольная точка

1) Контрольная точка выполняется периодически, посмотрим во втором терминале, какой интервал установлен.

```
postgres=# SHOW checkpoint_timeout;
checkpoint_timeout
-----
5min
(1 строка)
```

2) Контрольную точку можно запустить вручную.

```
postgres=# CHECKPOINT;
CHECKPOINT
```

Часть 6. Восстановление после сбоя

1) Добавим во втором терминале новые строки:

```
postgres=# INSERT INTO a SELECT id FROM generate_series(1,10000) AS id;
INSERT 0 10000
```

2) Остановите кластер БД в режиме системного сбоя. Для начала определим PID процесса `postmaster`.

```
astra@tantor:~$ sudo cat /var/lib/postgresql/tantor-se-17/data/postmaster.pid
12563
/var/lib/postgresql/tantor-se-17/data
1713849023
5432
/var/run/postgresql
*
  1048641      24
ready
astra@tantor:~$ sudo kill -SIGQUIT 12563
```

3) Запустим экземпляр сервера.

```
astra@tantor:~$ sudo systemctl start tantor-se-server-17
```

Идет восстановление.

4) Во втором окне посмотрим, сохранились ли вставленные строки.

```
postgres=# \c
```

Вы подключены к базе данных "postgres" как пользователь "postgres".

```
postgres=# SELECT count(*) FROM a;
count
-----
20001
(1 строка)
```

5) Очистим объекты во втором терминале.

```
postgres=# DROP TABLE a;
DROP TABLE
```

```
postgres=# \dt
```

Таблицы не найдены.

Глава 2b. Многоверсионность

Часть 1. Вставка, обновление и удаление строки

1) Запустите psql:

```
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
```

```
postgres=#
```

2) Создадим произвольную таблицу.

```
postgres=# CREATE TABLE a(id integer);
CREATE TABLE
```

3) Посмотрим, что получилось.

```
postgres=# \dt a
          Список отношений
 Схема | Имя | Тип      | Владелец
-----+-----+-----+-----
 public | a   | таблица | postgres
(1 строка)
```

4) Вставим первую строку в таблицу.

```
postgres=# INSERT INTO a VALUES (100);
INSERT 0 1
```

5) Посмотрим, какой номер транзакции xmin.

```
postgres=# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
-----+-----+-----
 1567 |    0 | 100
(1 строка)
```

Получился **1567** - это номер транзакции, в которой была создана первая версия строки.

6) Начнем явную транзакцию.

```
postgres=# BEGIN;
BEGIN
```

7) Обновим первую строчку.

```
postgres=# UPDATE a SET id = 200;
UPDATE 1
```

8) Обратимся и посмотрим, что получилось.

```
postgres=# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
-----+-----+-----
 1569 |    0 | 200
(1 строка)
```

9) Убедились в том, что транзакция видит свои изменения.

Как вы думаете, что будет если обратиться в параллельной транзакции?

id=100 или 200?

Во втором терминале обращаемся к таблице.

10) Запустите `psql`.

```
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
postgres=#
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1568 | 1569 | 100
(1 строка)
```

Обратите внимание, что `xmax` изменился - это значит, что уже существует вторая версия строки, но она еще не зафиксирована.

11) В первом терминале фиксируем транзакцию:

```
postgres=# COMMIT;
COMMIT
```

12) Во втором терминале теперь видим вторую строку.

```
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1569 |    0 | 200
(1 строка)
```

13) Теперь посмотрим, как выглядит удаление. Откроем транзакцию в первом терминале:

```
postgres=# BEGIN;
BEGIN
```

14) Удаляем строчку.

```
postgres=# DELETE FROM a;
DELETE 1
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
(0 rows)
```

Первая транзакция не видит строчку, она удалена, но изменение пока не зафиксировано.

15) Во втором терминале:

```
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1569 | 1570 | 200
(1 строка)
```

Строка еще видна, но `xmax` опять изменился.

16) В первом терминале фиксируем транзакцию:

```
postgres=# COMMIT;
COMMIT
```

17) Во втором терминале теперь видим изменение:

```
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
(0 rows)
```

Часть 2. Видимость версии строки на различных уровнях изоляции

1) Откроем первую транзакцию и вставим строку:

```
postgres=# BEGIN;
BEGIN
```

2) Посмотрим уровень изоляции:

```
postgres=# SHOW transaction_isolation;
transaction_isolation
-----
read committed
(1 строка)
```

```
postgres=# INSERT INTO a VALUES (100);
INSERT 0 1
```

```
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1571 |    0 | 100
(1 строка)
```

3) Начнем вторую транзакцию во втором терминале и обратимся к таблице:

```
postgres=# BEGIN;
BEGIN
```

```
postgres=# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
-----+-----+-----
(0 строк)
```

4) Посмотрим уровень изоляции:

```
postgres=# SHOW transaction_isolation;

transaction_isolation
-----
read committed
(1 строка)
```

5) Пока новая строка не видна. Зафиксируем первую транзакцию:

```
postgres=# COMMIT;
COMMIT
```

6) Во втором окне повторно обратимся к таблице. Что увидим?

```
postgres=# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
-----+-----+-----
 1571 |    0 | 100
(1 строка)
```

7) Зафиксируем вторую транзакцию:

```
postgres=# COMMIT;
```

COMMIT

Изменения стали видны. Это и есть аномалия неповторяющегося чтения.

Теперь в первом окне начнем транзакцию на уровне repeatable read.

8) Вставим еще одну строку:

```
postgres=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
```

```
postgres=*# INSERT INTO a VALUES (200);
INSERT 0 1
```

```
postgres=*# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1571 |    0 | 100
 1572 |    0 | 200
(2 строки)
```

9) Во второй транзакции обратимся к таблице в новой транзакции на том же уровне.

```
postgres=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
```

```
postgres=*# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1571 |    0 | 100
(1 строка)
```

10) Теперь фиксируем первую транзакцию:

```
postgres=*# COMMIT;
COMMIT
```

11) Обратимся ко второй транзакции еще раз:

```
postgres=*# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1571 |    0 | 100
(1 строка)
```

Изменения не видны. На этом уровне операторы транзакции работают только с одним снимком данных.

12) Зафиксируем вторую транзакцию:

```
postgres=*# COMMIT;
COMMIT
```

Часть 3. Состояние транзакции по CLOG

1) Откроем первую транзакцию и посмотрим после вставки состояние:

```
postgres=# BEGIN;
BEGIN
```

```
postgres=*# INSERT INTO a VALUES (300);
INSERT 0 1
```

```
postgres=*# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
```

```

1571 |    0 | 100
1572 |    0 | 200
1573 |    0 | 300
    
```

(3 строки)

2) Видим вставку третьей строки. Посмотрим статус транзакции:

```

postgres=# SELECT pg_xact_status('1573');
pg_xact_status
    
```

```

in progress
(1 строка)
    
```

3) Зафиксируем транзакцию и посмотрим статус:

```

postgres=# COMMIT;
COMMIT
    
```

```

postgres=# SELECT pg_xact_status('1573');
pg_xact_status
    
```

```

committed
(1 строка)
    
```

4) Теперь посмотрим, как поведет себя CLOG при откате транзакции:

```

postgres=# BEGIN;
BEGIN
    
```

```

postgres=# INSERT INTO a VALUES(400);
INSERT 0 1
    
```

```

postgres=# SELECT xmin, xmax, * FROM a;
    
```

```

xmin | xmax | id
-----+-----+-----
1571 |    0 | 100
1572 |    0 | 200
1573 |    0 | 300
1574 |    0 | 400
    
```

(4 строки)

```

postgres=# SELECT pg_xact_status('1574');
pg_xact_status
    
```

```

in progress
(1 строка)
    
```

```

postgres=# ROLLBACK;
ROLLBACK
    
```

```

postgres=# SELECT pg_xact_status('1574');
pg_xact_status
    
```

```

aborted
(1 строка)
    
```

```

postgres=# SELECT xmin, xmax, * FROM a;
    
```

```

xmin | xmax | id
-----+-----+-----
1571 |    0 | 100
1572 |    0 | 200
1573 |    0 | 300
    
```

(3 строки)

Часть 4. Блокировки таблицы

1) В первой транзакции вставим новую строку и посмотрим блокировки с помощью

`pg_locks`, для этого нам нужен `pid` обслуживающего процесса:

```
postgres=# SELECT pg_backend_pid();
pg_backend_pid
-----
          12193
```

(1 строка)

2) Откроем транзакцию и обратимся к таблице:

```
postgres=# BEGIN;
BEGIN
```

```
postgres=# UPDATE a SET id = id + 1;
UPDATE 3
```

```
postgres=# SELECT locktype, transactionid, mode, relation::regclass as obj FROM
pg_locks where pid = 12193;
```

locktype	transactionid	mode	obj
relation		AccessShareLock	pg_locks
relation		RowExclusiveLock	a
virtualxid		ExclusiveLock	
transactionid	1577	ExclusiveLock	

(4 строки)

Появилась блокировка на уровне таблицы `RowExclusiveLock` - накладывается в случае обновления строк.

3) Во втором окне построим индекс по таблице, предварительно посмотрим `pid`

процесса.

```
postgres=# SELECT pg_backend_pid();
pg_backend_pid
-----
          17210
```

(1 строка)

```
postgres=# CREATE INDEX ON a (id);
```

4) Транзакция подвисла. В первом терминале посмотрим, что происходит во втором процессе.

```
postgres=# SELECT locktype, transactionid, mode, relation::regclass as obj FROM
pg_locks where pid = 17210;
```

locktype	transactionid	mode	obj
virtualxid		ExclusiveLock	
relation		ShareLock	a

(2 строки)

Появилась блокировка `ShareLock`, она не совместима с `RowExclusiveLock`, возникла блокировочная ситуация.

5) Зафиксируем первую транзакцию:

```
postgres=# # COMMIT;
COMMIT
```

6) Тут же срабатывает команда во втором окне:

```
CREATE INDEX
```

Часть 5. Блокировка строки

1) Начнем первую транзакцию:

```
postgres=# BEGIN;
BEGIN
```

```
postgres=# UPDATE a SET id = id + 1 WHERE id=101;
UPDATE 1
```

2) Начнем вторую транзакцию:

```
postgres=# BEGIN;
BEGIN
postgres=# UPDATE a SET id = id + 1 WHERE id=101;
```

Транзакция подвисла, сработала блокировка.

3) Зафиксируем первую транзакцию:

```
postgres=# COMMIT;
COMMIT
```

Тут же срабатывает вторая.

```
UPDATE 0
postgres=# COMMIT;
COMMIT
```

Обратите внимание, что обновление не произошло, теперь такой версии строки нет для обновления.

4) В первом терминале обратимся к таблице:

```
postgres=# SELECT xmin, xmax, * FROM a;
```

```
xmin | xmax | id
-----+-----+-----
 1577 |    0 | 201
 1577 |    0 | 301
 1579 | 1580 | 102
(3 строки)
```

5) Удалим таблицу:

```
postgres=# DROP TABLE a;
DROP TABLE
```

Глава 2с. Регламентные работы

Часть 1. Обычная очистка таблицы

1) Запустите psql:

```

astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
postgres=#
    
```

2) Создадим произвольную таблицу:

```

postgres=# CREATE TABLE a (id integer primary key generated always as identity, t
char(2000)) WITH (autovacuum_enabled = off);
CREATE TABLE
    
```

```

postgres=# INSERT INTO a(t) SELECT to_char(generate_series(1,10000), '9999');
INSERT 0 10000
    
```

3) Посмотрим, что получилось:

```

postgres=# \d a
                Table "public.a"
Column |      Type      | Collation | Nullable |      Default
-----+-----+-----+-----+-----
id     | integer       |           | not null | generated always as identity
t      | character(2000) |           |         |
Indexes:
  "a_pkey" PRIMARY KEY, btree (id)
    
```

Обратите внимание: создан первичный ключ и индекс.

4) Узнаем размер таблицы и индекса в байтах:

```

postgres=# SELECT pg_table_size('a');
pg_table_size
-----
      20512768
(1 строка)
    
```

```

postgres=# SELECT pg_table_size('a_pkey');
pg_table_size
-----
      245760
(1 строка)
    
```

5) Обновим 50% строк:

```

postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000
    
```

6) Посмотрим размеры объектов:

```

postgres=# SELECT pg_table_size('a');
pg_table_size
-----
     30752768
(1 строка)
    
```

```

postgres=# SELECT pg_table_size('a_pkey');
pg_table_size
-----
    
```

360448

(1 строка)

7) Они также увеличились. Очистим таблицу и индекс:

```
postgres=# VACUUM a;
VACUUM
```

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
```

```
-----
30760960
```

(1 строка)

```
pg_table_size
```

```
-----
360448
```

(1 строка)

8) Размер остался таким же. Еще раз обновим строки:

```
postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000
```

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
```

```
-----
30760960
```

(1 строка)

```
pg_table_size
```

```
-----
360448
```

(1 строка)

Опять размер не изменился. Произошло это потому, что было использовано очищенное пространство.

9) К примеру предположим, что пропущен цикл очистки:

```
postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000
```

```
postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000
```

```
postgres=# SELECT pg_table_size('a');
SELECT pg_table_size('a_pkey');
pg_table_size
```

```
-----
51249152
```

(1 строка)

```
pg_table_size
```

```
-----
466944
```

(1 строка)

10) Размер объектов опять вырос:

```
postgres=# VACUUM a;
VACUUM
```

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
```

```
51249152
(1 строка)
```

```
pg_table_size
-----
      466944
(1 строка)
```

Даже после очистки размер не уменьшается.

Часть 2. Анализ таблицы

1) Так как произошло несколько циклов обновлений, посмотрим насколько актуальна осталась статистика. Сначала обратимся к системному каталогу:

```
postgres=# SELECT reltuples FROM pg_class WHERE relname='a';
reltuples
-----
      8333
(1 строка)
```

Получили, что в таблице у нас содержится 8333 строки.

2) Теперь обратимся к таблице:

```
postgres=# SELECT count(*) FROM a;
count
-----
 10000
(1 строка)
```

3) Оказалось, что строк больше. Статистика всегда приближительна. Вызовем вторую фазу анализа:

```
postgres=# ANALYZE a;
ANALYZE
```

4) Теперь статистика стала более точной:

```
postgres=# SELECT reltuples FROM pg_class WHERE relname='a';

reltuples
-----
      10000
(1 строка)
```

Часть 3. Перестройка индекса

1) Посмотрим, какой размер объектов:

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
-----
      51249152
(1 строка)
```

```
pg_table_size
-----
      466944
(1 строка)
```

2) Сейчас в таблице а один только индекс. Перестроим его:

```
postgres=# REINDEX TABLE a;
REINDEX
```

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
-----
      51249152
(1 строка)
```

```
pg_table_size
-----
      245760
(1 строка)
```

3) Размер индекса уменьшился, размер таблицы остался неизменным.

Часть 4. Полная очистка

```
postgres=# VACUUM FULL a;
VACUUM
```

1) Посмотрим размер объектов:

```
postgres=# SELECT pg_table_size('a');
SELECT pg_table_size('a_pkey');
```

```
pg_table_size
-----
      20488192
(1 строка)
```

```
pg_table_size
-----
      245760
(1 строка)
```

Размер таблицы уменьшился.

2) Удалим таблицу:

```
postgres=# DROP TABLE a;
DROP TABLE
```

Задание выполнено.

Часть 5. Расширение HypoPG

1) Установите расширение hypoPg:

```
postgres=# CREATE EXTENSION hypoPg;
CREATE EXTENSION
```

2) Создайте таблицу с тестовыми данными:

```
postgres=# CREATE TABLE hypo AS SELECT id, 'line ' || id AS val FROM
generate_series(1,10000) id;
SELECT 10000
```

3) План выполнения выборки одной строки - последовательное сканирование (Seq Scan).

Индексных методов доступа нет, так как нет индексов:

```
postgres=# EXPLAIN SELECT * FROM hypo WHERE id = 1;
          QUERY PLAN
```

```
-----
Seq Scan on hypo (cost=0.00..165.60 rows=41 width=36)
  Filter: (id = 1)
(2 строки)
```

Почему ожидаемое количество строк 41, а не 1? Нет статистики.

4) Соберите статистику:

```
postgres=# vacuum analyze hypo;
VACUUM
postgres=# EXPLAIN SELECT * FROM hypo WHERE id = 1;
          QUERY PLAN
```

```
-----
Seq Scan on hypo (cost=0.00..180.00 rows=1 width=13)
  Filter: (id = 1)
(2 строки)
```

Ожидаемое количество строк 1.

Дана задача оптимизировать выполнение этого запроса. Предполагаем, что индекс по столбцу `id` ускорит выполнение запроса. Нужно убедиться, что планировщик будет использовать индекс. Если планировщик не будет использовать индекс, то предположение неверно и индекс создавать не нужно. Создание индекса трудоемко и долго, он занимает место. Перед созданием индекса мы хотим проверить гипотезу о том, что планировщик его будет использовать при выполнении оптимизируемого запроса.

5) Для проверки гипотезы создайте гипотетический индекс:

```
postgres=# SELECT * FROM hypopg_create_index('CREATE INDEX hypo_idx ON hypo (id)');
 indexrelid |      indexname
-----+-----
    13495 | <13495>btree_hypo_id
(1 строка)
```

Имя гипотетического индекса генерируется автоматически, это нормально.

Реальный индекс не создаётся, команда выполняется моментально.

6) Посмотрите список гипотетических индексов:

```
postgres=# SELECT * FROM hypopg_list_indexes;
 indexrelid |      index_name      | schema_name | table_name | am_name
-----+-----+-----+-----+-----
    13495 | <13495>btree_hypo_id | public      | hypo      | btree
(1 строка)
```

Какой план выполнения сейчас?

7) Выполните команду:

```
postgres=# EXPLAIN SELECT * FROM hypo WHERE id = 1;
          QUERY PLAN
-----
Index Scan using "<13495>btree_hypo_id" on hypo (cost=0.04..8.05 rows=1 width=13)
  Index Cond: (id = 1)
(2 строки)
```

План показывает, что индекс будет использоваться.

Реального индекса нет, поэтому план реального выполнения использует сканирование

таблицы:

```
postgres=# EXPLAIN (analyze) SELECT * FROM hypo WHERE id = 1;
               QUERY PLAN
```

```
-----
Seq Scan on hypo (cost=0.00..180.00 rows=1 width=13) (actual time=0.025..0.875 rows=1
loops=1)
  Filter: (id = 1)
  Rows Removed by Filter: 9999
  Planning Time: 0.077 ms
  Execution Time: 1.074 ms
(5 строк)
```

8) Создайте реальный индекс:

```
postgres=# CREATE UNIQUE INDEX hypo_id ON hypo(id);
CREATE INDEX
```

План выполнения остался прежним:

```
postgres=# EXPLAIN SELECT * FROM hypo WHERE id = 1;
               QUERY PLAN
```

```
-----
Index Scan using "<13495>btree_hypo_id" on hypo (cost=0.04..8.05 rows=1 width=13)
  Index Cond: (id = 1)
(2 строки)
```

9) Уберите побочные эффекты:

```
postgres=# SELECT hypopg_reset();
 hypopg_reset
```

```
-----
(1 строка)
```

Планировщик стал использовать созданный индекс:

```
postgres=# EXPLAIN SELECT * FROM hypo WHERE id = 1;
               QUERY PLAN
```

```
-----
Index Scan using hypo_id on hypo (cost=0.29..8.30 rows=1 width=13)
  Index Cond: (id = 1)
(2 строки)
```

Расширение позволяет скрывать от планировщика реальные индексы:

```
postgres=# SELECT hypopg_hide_index('hypo_id'::regclass);
 hypopg_hide_index
```

```
-----
t
(1 строка)
```

Скрытие действует только в пределах сессии и на работу других сессий не оказывает влияние.

Гипотетические индексы также существуют только в рамках сессии.

Гипотетические индексы при этом исчезают:

```
postgres=# SELECT * FROM hypopg_list_indexes;
 indexrelid | index_name | schema_name | table_name | am_name
-----+-----+-----+-----+-----
(0 строк)
```

План выполнения будет использовать последовательное сканирование:

```
postgres=# EXPLAIN SELECT * FROM hypo WHERE id = 1;
```

QUERY PLAN

```
-----
Seq Scan on hypo (cost=0.00..180.00 rows=1 width=13)
  Filter: (id = 1)
(2 строки)
```

Есть представление со списком скрытых в данной сессии индексов:

```
postgres=# SELECT * FROM hypopg_hidden_indexes;
 indexrelid | index_name | schema_name | table_name | am_name | is_hypo
-----+-----+-----+-----+-----+-----
      17402 | hypo_id   | public      | hypo       | btree   | f
(1 строка)
```

10) Убедитесь, что скрытие индексов и гипотетические индексы существуют только в рамках сессии:

```
postgres=# SELECT * FROM hypopg_create_index('CREATE INDEX hypo_idx ON hypo (id)');
 indexrelid |      indexname
-----+-----
      13495 | <13495>btree_hypo_id
(1 строка)
```

```
postgres=# SELECT * FROM hypopg_list_indexes;
 indexrelid |      index_name      | schema_name | table_name | am_name
-----+-----+-----+-----+-----
      13495 | <13495>btree_hypo_id | public      | hypo       | btree
(1 строка)
```

```
postgres=# \q
postgres@tantor:~$ psql
psql (17.5)
```

11) Введите "help", чтобы получить справку:

```
postgres=# SELECT * FROM hypopg_list_indexes;
 indexrelid | index_name | schema_name | table_name | am_name
-----+-----+-----+-----+-----
(0 строк)
```

```
postgres=# SELECT * FROM hypopg_hidden_indexes;
 indexrelid | index_name | schema_name | table_name | am_name | is_hypo
-----+-----+-----+-----+-----+-----
(0 строк)
```

Глава 2d. Выполнение запросов

Часть 1. Создание объектов для запросов

1) Запустите psql:

```
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
postgres=#
```

2) Создадим новую таблицу и заполним данными:

```
postgres=# CREATE TABLE test (col1 integer, col2 integer, name text);
CREATE TABLE
postgres=# INSERT INTO test VALUES (1,2,'test1');
INSERT 0 1
postgres=# INSERT INTO test VALUES (3,4,'test2');
INSERT 0 1
```

3) Создадим представление над таблицей:

```
postgres=# CREATE VIEW v_table AS
      SELECT * FROM test;
CREATE VIEW
postgres=# SELECT col1, col2 FROM v_table WHERE name='test1'::text ;
 col1 | col2
-----+-----
     1 |     2
(1 строка)
```

Часть 2. Последовательное чтение блоков таблицы (Seq Scan)

1) С помощью команды Explain посмотрим план выполнения запроса:

```
postgres=# EXPLAIN SELECT col1, col2 FROM v_table WHERE name='test1'::text ;
          QUERY PLAN
-----
Seq Scan on test  (cost=0.00..25.00 rows=6 width=8)
  Filter: (name = 'test1'::text)
(2 строки)
```

Видим, что использовалось последовательное чтение таблицы `test`. То есть представление было раскрыто, данные извлечены непосредственно с таблицы.

2) Применим параметры `analyze` и `buffers`. Они показывают, что запрос был выполнен реально, и какое количество страниц было затронуто.

```
postgres=# EXPLAIN(analyze, buffers, costs off, timing off)
SELECT col1, col2 FROM v_table WHERE name='test1'::text ;
          QUERY PLAN
-----
Seq Scan on test  (actual rows=1 loops=1)
  Filter: (name = 'test1'::text)
  Rows Removed by Filter: 1
  Buffers: shared read=1
Planning Time: 0.063 ms
Execution Time: 9.569 ms
(6 строк)
```

Часть 3. Возвращение данных по индексу

1) Создадим индекс по столбцу `col1`:

```
postgres=# CREATE INDEX ON test (col1);
CREATE INDEX
```

```
postgres=# \d test
```

```

                Таблица "public.test"
  Столбец |  Тип  | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----
 col1    | integer |                    |                    |
 col2    | integer |                    |                    |
 name    | text    |                    |                    |

```

Индексы:

```
"test_col1_idx" btree (col1)
```

2) Можно убедиться, что формирование имени индекса производится автоматически, добавим информации к таблице:

```
postgres=# INSERT INTO test(col1,col2)
          SELECT generate_series(3,1003), generate_series(4,1004);
INSERT 0 1001
```

3) Посмотрим, что получится, если будем выбирать малое количество строк. То есть, случай, когда будет высокая селективность и маленькая кардинальность:

```
postgres=# EXPLAIN(analyze, buffers, costs off, timing off)
SELECT col1, col2 FROM test WHERE col1<20;
```

```

                QUERY PLAN
-----
Index Scan using test_col1_idx on test (actual rows=19 loops=1)
  Index Cond: (col1 < 20)
  Buffers: shared hit=3
  Planning:
    Buffers: shared hit=17
    Planning Time: 0.179 ms
    Execution Time: 0.117 ms
(7 строк)

```

Убедились, что используется индексный доступ.

Часть 4. Низкая селективность

Теперь отберем большое количество строк:

```
postgres=# SELECT count(*) FROM test;
count
-----
  1003
(1 строка)
```

Всего строк 1003

```
postgres=# EXPLAIN(analyze, buffers, costs off, timing off)
SELECT col1, col2 FROM test WHERE col1>20;
```

```

                QUERY PLAN
-----
Seq Scan on test (actual rows=983 loops=1)
  Filter: (col1 > 20)
  Rows Removed by Filter: 20
  Buffers: shared hit=5
  Planning:
    Buffers: shared hit=3

```

Planning Time: 0.157 ms
 Execution Time: 0.201 ms
 (8 строк)

Отобрано 983 строки. Что означает низкую селективность и высокую кардинальность.

Убедились, что в этом случае индексный доступ становится дорогим, и PostgreSQL переходит к последовательному доступу.

Часть 5. Использование статистики

К примеру, при заполнении таблицы `test` третий столбец был не заполнен. Давайте посмотрим какой процент будет иметь значение NULL

Пересоберем статистику:

```
postgres=# ANALYZE test;
ANALYZE
```

```
postgres=# SELECT stanullfrac FROM pg_statistic WHERE starelid = 'test'::regclass AND
staattnum = 3;
stanullfrac
-----
 0.9981884
(1 row)
```

Значение NULL в более, чем 99% строк.

Часть 6. Представление pg_stat_statements

1) Убедимся, что представление установлено:

```
postgres=# \dx pg_stat_statements
                Список установленных расширений
  Имя           | Версия | Схема | Описание
-----+-----+-----+-----
pg_stat_statements| 1.10   | public | track planning and execution statistics of all SQL statements executed
(1 строка)
```

2) Посмотрим, какие столбцы есть в представлении.

```
postgres=# \d pg_stat_statements
Представление "public.pg_stat_statements"
  Столбец          | Тип                | Правило сортировки | Допустимость NULL |
-----+-----+-----+-----
userid            | oid                |                     |                     |
dbid              | oid                |                     |                     |
toplevel          | boolean            |                     |                     |
queryid           | bigint             |                     |                     |
query             | text               |                     |                     |
plans             | bigint             |                     |                     |
total_plan_time   | double precision   |                     |                     |
min_plan_time     | double precision   |                     |                     |
max_plan_time     | double precision   |                     |                     |
mean_plan_time    | double precision   |                     |                     |
stddev_plan_time  | double precision   |                     |                     |
calls             | bigint             |                     |                     |
total_exec_time   | double precision   |                     |                     |
min_exec_time     | double precision   |                     |                     |
max_exec_time     | double precision   |                     |                     |
mean_exec_time    | double precision   |                     |                     |
stddev_exec_time  | double precision   |                     |                     |
rows              | bigint             |                     |                     |
shared_blks_hit   | bigint             |                     |                     |
shared_blks_read  | bigint             |                     |                     |
shared_blks_dirtied | bigint             |                     |                     |
```

shared_blks_written	bigint			
local_blks_hit	bigint			
local_blks_read	bigint			
local_blks_dirtied	bigint			
local_blks_written	bigint			
temp_blks_read	bigint			
temp_blks_written	bigint			
blk_read_time	double precision			
blk_write_time	double precision			
temp_blk_read_time	double precision			
temp_blk_write_time	double precision			
wal_records	bigint			
wal_fpi	bigint			
wal_bytes	numeric			
jit_functions	bigint			
jit_generation_time	double precision			
jit_inlining_count	bigint			
jit_inlining_time	double precision			
jit_optimization_count	bigint			
jit_optimization_time	double precision			
jit_emission_count	bigint			
jit_emission_time	double precision			

3) Сбросим статистику, которую собирает расширение:

```
postgres=# SELECT pg_stat_statements_reset();
pg_stat_statements_reset
```

(1 строка)

4) Обратимся к таблице test:

```
postgres=# EXPLAIN (analyze)
SELECT col1, col2 FROM test WHERE col1>20;
QUERY PLAN
```

```
Seq Scan on test (cost=0.00..17.54 rows=984 width=8) (actual time=0.022..0.132 rows=983 loops=1)
  Filter: (col1 > 20)
  Rows Removed by Filter: 20
Planning Time: 0.190 ms
Execution Time: 0.234 ms
(5 строк)
```

5) С помощью представления pg_stat_statements посмотрим, сколько времени заняло выполнение запроса и сколько страниц было задействовано:

```
postgres=# SELECT queryid, substring(query FOR 100) as query, total_exec_time as ms,
shared_blks_hit as blocks
from pg_stat_statements
WHERE query LIKE '%col1, col2%';
```

queryid	query	ms	blocks
-3250261183448805182	EXPLAIN (analyze) SELECT col1, col2 FROM test WHERE col1>\$1	0.491265	11

(1 строка)

Глава 2е. Расширения

Часть 1. Определение директории с файлами расширения

1) Перейдем под пользователя postgres:

```
astra@tantor:~$ sudo su - postgres
```

2) В командной строке воспользуемся утилитой pg_config:

```
postgres@education:~$ pg_config --sharedir
```

```
/opt/tantor/db/17/share/postgresql
```

3) Добавим к получившемуся пути extension:

```
postgres@education:~$ ls -l /opt/tantor/db/17/share/postgresql/extension/
```

```
...
-rw-r--r-- 1 root root 31171 vector--0.8.0.sql
-rw-r--r-- 1 root root 145 vector.control
-rw-r--r-- 1 root root 944 xml2--1.0--1.1.sql
-rw-r--r-- 1 root root 2049 xml2--1.1.sql
-rw-r--r-- 1 root root 182 xml2.control
```

4) Запустите psql:

```
postgres@tantor:~$ psql
```

```
psql (17.5)
```

Введите "help", чтобы получить справку.

```
postgres=#
```

5) Определим путь расширения с помощью функции pg_config():

```
postgres=# SELECT setting FROM pg_config()
where name = 'SHAREDIR';
```

```
setting
```

```
-----
/opt/tantor/db/17/share/postgresql
```

```
(1 row)
```

Часть 2. Просмотр установленных расширений

```
postgres=# \dx
```

Список установленных расширений

Имя	Версия	Схема	Описание
pg_stat_statements	1.10	public	track planning and execution statistics of all SQL
pg_store_plans	1.6.4	public	track plan statistics of all SQL statements executed
plpgsql	1.0	pg_catalog	PL/pgSQL procedural language

(3 строки)

Часть 3. Просмотр доступных расширений

Воспользуемся расширением pg_available_extensions:

```
postgres=# SELECT * from pg_available_extensions;
```

name	default_version	installed_ver	comment
...			
pg_columnar	11.1-12		Hydra Columnar extension
pgrowlocks	1.2		show row-level locking information
pg_ivm	1.10		incremental view maintenance on PostgreSQL
pg_freespacem	1.2		examine the free space map (FSM)
plperl	1.0		PL/Perl procedural language
pg_store_plans	1.8.1		track plan statistics of all SQL statements executed

```

pg_buffercache | 1.5 | | examine the shared buffer cache
refint | 1.0 | | functions for implementing referential integrity (obsolete)
jsonb_plpython3u | 1.0 | | transform between jsonb and plpython3u
pgaudit | 17.0 | | provides auditing functionality
pltcl | 1.0 | | PL/Tcl procedural language
plperl | 1.0 | | PL/PerlU untrusted procedural language
pg_throttle | 1.1.0 | | throttle resources usage
(91 строка)
    
```

В примере доступно 91 расширение.

Часть 4. Установка и удаление произвольного обновления

1) Например, установим расширение `pg_surgery`:

```

postgres=# CREATE EXTENSION pg_surgery;
CREATE EXTENSION
    
```

```

postgres=# \dx
    
```

```

          Список установленных расширений
-----+-----+-----+-----+
      Имя | Версия | Схема | Описание
-----+-----+-----+-----+
pg_stat_statements | 1.10 | public | track planning and execution statistics of all SQL
pg_store_plans | 1.6.4 | public | track plan statistics of all SQL statements executed
pg_surgery | 1.0 | public | extension to perform surgery on a damaged relation
plpgsql | 1.0 | pg_catalog | PL/pgSQL procedural language
(4 строки)
    
```

2) Посмотрим содержимое расширения:

```

postgres=# \dx+ pg_surgery
      Объекты в расширении "pg_surgery"
      Описание объекта
-----+-----+-----+-----+
функция heap_force_freeze(regclass,tid[])
функция heap_force_kill(regclass,tid[])
(2 строки)
    
```

3) Удалим расширение:

```

postgres=# DROP EXTENSION pg_surgery;
DROP EXTENSION
    
```

Часть 5. Просмотр доступных версий расширений. Обновление до актуальной версии

1) Воспользуемся представлением `pg_available_extension_versions`:

```

postgres=# SELECT name, version FROM pg_available_extension_versions WHERE name =
'pg_stat_kcache';
      name | version
-----+-----+
pg_stat_kcache | 2.1.2
pg_stat_kcache | 2.2.2
pg_stat_kcache | 2.1.1
pg_stat_kcache | 2.2.0
pg_stat_kcache | 2.2.1
pg_stat_kcache | 2.1.0
pg_stat_kcache | 2.3.0
pg_stat_kcache | 2.2.3
pg_stat_kcache | 2.1.3
(9 rows)
    
```

2) Для начала установим версию 2.2.2:

```

postgres=# CREATE EXTENSION pg_stat_kcache VERSION '2.2.2';
ERROR:  required extension "pg_stat_statements" is not installed
    
```

```
HINT: Use CREATE EXTENSION ... CASCADE to install required extensions too.
postgres=# CREATE EXTENSION pg_stat_kcache VERSION '2.2.2' cascade;
NOTICE: installing required extension "pg_stat_statements"
ERROR: This module can only be loaded via shared_preload_libraries
postgres=# show shared_preload_libraries ;
shared_preload_libraries
-----
```

(1 row)

Для загрузки библиотек их нужно указать в параметре конфигурации. Если библиотеки `pg_stat_statements`, `pg_stat_kcache` не загружены, то выполните команду:

```
postgres=#
alter system set shared_preload_libraries = pg_stat_statements, pg_stat_kcache ;
ALTER SYSTEM
```

После запятой пробел обязателен.

Изменение этого параметра требует перезапуск экземпляра. Остановите экземпляра

утилитой `pg_ctl` и запустите его заново как службу:

```
postgres=# \q
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
postgres@tantor:~$ psql
```

3) Установите расширение:

```
postgres=# CREATE EXTENSION pg_stat_kcache VERSION '2.2.2' cascade;
NOTICE: installing required extension "pg_stat_statements"
CREATE EXTENSION
postgres=# \dx pg_stat*
```

```

                List of installed extensions
   Name          | Version | Schema | Description
-----+-----+-----+-----
 pg_stat_kcache  | 2.2.2   | public | Kernel statistics gathering
 pg_stat_statements | 1.11    | public | track planning and execution statistics
                                     of all SQL statements executed
```

(2 rows)

4) Посмотрим, можно ли расширение `pg_stat_kcache` обновить до версии 2.3.0.

Воспользуемся функцией `pg_extension_update_paths`:

```
postgres=# SELECT * FROM pg_extension_update_paths('pg_stat_kcache') where source
= '2.2.2' and target='2.3.0';
 source | target | path
-----+-----+-----
 2.2.2  | 2.3.0  | 2.2.2--2.2.3--2.3.0
(1 row)
```

5) Обновим расширение до последней версии:

```
postgres=# ALTER EXTENSION pg_stat_kcache UPDATE;
ERROR: cannot drop view pg_stat_kcache_detail because other objects depend on it
DETAIL: view pg_stat_kcache depends on view pg_stat_kcache_detail
HINT: Use DROP ... CASCADE to drop the dependent objects too.
postgres=# ALTER EXTENSION pg_stat_kcache UPDATE TO "2.2.3";
ALTER EXTENSION
```

Мы нашли ошибку в скрипте обновления `pg_stat_kcache--2.2.3--2.3.0.sql`. Удалим

расширение и установим **последнюю** версию:

```
postgres=# drop extension pg_stat_kcache;
DROP EXTENSION
postgres=# create extension pg_stat_kcache;
CREATE EXTENSION
```

```
postgres=# \dx pg_stat_kcache
                List of installed extensions
  Name          | Version | Schema | Description
-----+-----+-----+-----
 pg_stat_kcache | 2.3.0   | public | Kernel statistics gathering
(1 row)
```

6) Удалим расширение:

```
postgres=# DROP EXTENSION pg_stat_kcache;
DROP EXTENSION
```

Часть 6. Обертки внешних данных

1) Посмотрим, какие есть обертки внешних данных (FDW):

```
postgres=# SELECT * FROM pg_available_extensions
WHERE name LIKE '%fdw%';
 name          | default_version | installed_version | comment
-----+-----+-----+-----
 postgres_fdw | 1.1              |                   | foreign-data wrapper for remote PostgreSQL servers
 file_fdw      | 1.0              |                   | foreign-data wrapper for flat file access
(2 строки)
```

2) Воспользуемся оберткой внешней данных для подключения к PostgreSQL:

```
postgres=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

```
postgres=# \dx postgres_fdw
                Список установленных расширений
  Имя          | Версия | Схема | Описание
-----+-----+-----+-----
 postgres_fdw | 1.1    | public | foreign-data wrapper for remote PostgreSQL
 servers
(1 строка)
```

3) Посмотрим, какие есть базы данных:

```
postgres=# \l
                Список баз данных
  Имя          | Владелец | Кодировка | Провайдер локали | LC_COLLATE | LC_CTYPE | локаль ICU | Правила ICU | Права доступа
-----+-----+-----+-----+-----+-----+-----+-----+-----
 postgres     | postgres | UTF8      | libc              | ru_RU.UTF-8 | ru_RU.UTF-8 |             |             | =c/postgres +
 template0    | postgres | UTF8      | libc              | ru_RU.UTF-8 | ru_RU.UTF-8 |             |             | =c/postgres +
 template1    | postgres | UTF8      | libc              | ru_RU.UTF-8 | ru_RU.UTF-8 |             |             | =c/postgres +
 test_db      | postgres | UTF8      | libc              | ru_RU.UTF-8 | ru_RU.UTF-8 |             |             | postgres=Ctc/postgres
(4 строки)
```

4) Подключимся и вернем информацию с БД test_db. Для начала создадим объект удаленного сервера:

```
postgres=# CREATE SERVER test FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host
'localhost', port '5432', dbname 'test_db');
```

CREATE SERVER

```
postgres=# \des
          Список сторонних серверов
Имя  | Владелец | Обёртка сторонних данных
-----+-----+-----
test | postgres | postgres_fdw
(1 строка)
```

5) После этого создадим пользователя, под которым будет происходить подключение.

Отображений на пользователя может быть несколько:

```
postgres=# CREATE USER MAPPING FOR postgres SERVER test
OPTIONS ( user 'postgres', password 'postgres' );
CREATE USER MAPPING
```

```
postgres=# \deu
Список сопоставлений пользователей
Сервер | Имя пользователя
-----+-----
test   | postgres
(1 строка)
```

6) После создадим таблицу, к которой можно будет подключиться:

```
postgres=# CREATE FOREIGN TABLE order_remote
( id bigint, name varchar(32)
server test
OPTIONS ( schema_name 'public', table_name 'order_items_1'
);
CREATE FOREIGN TABLE
```

```
postgres=# \det
Список сторонних таблиц
Схема | Таблица | Сервер
-----+-----+-----
public | order_remote | test
(1 строка)
```

7) Обращаемся к этой таблице, как к обычной таблице:

```
postgres=# SELECT * FROM order_remote LIMIT 10;
id | name
----+-----
0  |
1  |
2  |
3  |
4  |
5  |
6  |
7  |
8  |
9  |
(10 строк)
```

8) Описание удаленной таблицы можно получить, как обычной:

```
postgres=# \d order_remote
```

Сторонняя таблица "public.order_remote"

Столбец	Тип	Правило сортировки	Допустимость NULL	По умолчанию
id	bigint			
name	character varying(32)			

Сервер: test

Параметр ОВД: (schema_name 'public', table_name 'order_items_1')

9) Посмотрим, откуда приходят данные:

```
postgres=# EXPLAIN SELECT * FROM order_remote LIMIT 10;
          QUERY PLAN
```

```
-----
Foreign Scan on order_remote  (cost=100.00..100.42 rows=10 width=90)
(1 строка)
```

10) Очистим базу данных:

```
postgres=# DROP FOREIGN TABLE order_remote;
DROP FOREIGN TABLE
```

```
postgres=# DROP USER MAPPING FOR postgres server test;
DROP USER MAPPING
```

```
postgres=# DROP SERVER test;
DROP SERVER
```

```
postgres=# DROP EXTENSION postgres_fdw;
DROP EXTENSION
```

Глава 3. Конфигурирование

Часть 1. Обзор параметров конфигурации

1) Сколько параметров конфигурации имеется?

```
postgres=# select count(*) from pg_settings;
count
-----
    392
(1 строка)
```

2) Сколько имеется системных параметров? Выполните запрос:

```
postgres=# select count(name) from pg_settings where name not like '%.%';
count
-----
    392
(1 строка)
```

Параметры, в названии которых имеется **точка**, относятся к расширениям, библиотекам, приложениям (customized options, внесистемные параметры, пользовательские настройки) и их может быть произвольное количество. Загруженные модули могут регистрировать свои параметры конфигурации.

Для загрузки библиотек их нужно указать в параметре конфигурации. Выполните команду:

```
postgres=#
alter system set shared_preload_libraries = pg_stat_statements, pg_qualstats,
pg_store_plans, pg_prewarm, pg_stat_kcache;
ALTER SYSTEM
```

После запятой пробел обязателен.

Изменение этого параметра требует перезапуск экземпляра. Остановите экземпляр утилитой `pg_ctl` и запустите его заново как службу:

```
postgres=# \q
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
postgres@tantor:~$ psql
```

3) Какие библиотеки **были загружены**?

```
postgres=# show shared_preload_libraries;
shared_preload_libraries
-----
pg_stat_statements, pg_qualstats, pg_store_plans, pg_prewarm, pg_stat_kcache
(1 строка)
```

Было загружено пять библиотек.

4) Сколько параметров модулей (библиотек) и приложений есть? В названии таких параметров есть **точка**. Выполните запрос:

```
postgres=# select distinct split_part(name, '.', 1), count(name) from pg_settings
where name like '%.%' group by split_part(name, '.', 1) order by 1;
 split_part | count
-----+-----
pg_prewarm |      2
pg_qualstats |      8
pg_stat_kcache |      3
pg_stat_statements |      8
pg_store_plans |     15
(5 строк)
```

5) Какие **максимальные** значения есть у параметров? Это интересно, чтобы сопоставить название типа параметра с его размерностью (сколько байт занимает значение). Выполните запрос:

```
postgres=# select vartype, min_val, max_val, count(name) from pg_settings group
by vartype, min_val, max_val order by length(max_val) desc, vartype;
 vartype | min_val | max_val | count
-----+-----+-----+-----
bool | | | 122
enum | | | 44
string | | | 68
int64 | 0 | 9223372036854775807 | 1
int64 | 100000 | 9223372036854775807 | 1
int64 | -1 | 9223372036854775807 | 1
int64 | 10000 | 9223372036854775807 | 4
real | -1 | 1.79769e+308 | 3
real | 0 | 1.79769e+308 | 7
int64 | 0 | 21000000000 | 2
integer | 100 | 1073741823 | 2
integer | -1 | 2147483647 | 13
integer | 1 | 2147483647 | 6
integer | -2147483648 | 2147483647 | 1
integer | -1 | 1073741823 | 2
...
```

Для продолжения вывода можно нажать клавишу <z>:

Максимальное значение типа с названием `int64`: `9223372036854775807` = 2 в степени 63 минус 1, что соответствует максимуму для целочисленного знакового типа размерностью **64** бита.

У типов с названием `integer` максимальное значение - 2147483647, что соответствует максимуму для целочисленного знакового типа размерностью 32 бита.

6) **Контекст** указывает, можно ли изменить значение параметра, и если можно, то **каким способом**. Какие контексты параметров есть и сколько параметров в каждом контексте?

```
postgres=# select context, count(name) from pg_settings where name not like '%.%' group
by context order by 1;
 context | count
-----+-----
backend |      2
internal |     19
postmaster |     67
sighup |    100
superuser |     47
superuser-backend |      4
user |    153
(7 строк)
```

Больше всего параметров **контекста** `user`. Изменения параметров **контекста** `postmaster` потребуют рестарт экземпляра. Параметры **контекста** `internal` только для чтения (не могут меняться командами `SET`, `ALTER SYSTEM`, путем установки значения в файлах параметров конфигурации) и их нет смысла указывать в файлах параметров конфигурации. Могут ли поменяться значения параметров **контекста** `internal`? Могут. Способ изменения зависит от параметра. Например, значение параметра `wal_segment_size` может меняться утилитой `pg_resetwal`, параметра `data_checksums` - утилитой `pg_checksums`.

7) Посмотрите, какие категории параметров есть:

```
postgres=# select category, count(*) from pg_settings group by category order by 2 desc;
-----+-----
category                                     | count
-----+-----
Customized Options                          |    36
Client Connection Defaults / Statement Behavior |    33
Developer Options                           |    26
Resource Usage / Memory                     |    27
Query Tuning / Planner Method Configuration |    25
Reporting and Logging / What to Log         |    21
Preset Options                              |    19
Write-Ahead Log / Settings                  |    15
Connections and Authentication / SSL        |    14
Query Tuning / Planner Cost Constants       |    13
Reporting and Logging / Where to Log        |    13
Autovacuum                                  |    13
Client Connection Defaults / Locale and Formatting |    12
Connections and Authentication / Connection Settings |    11
Replication / Standby Servers               |    11
Resource Usage / Asynchronous Behavior     |     9
Write-Ahead Log / Recovery Target          |     8
Query Tuning / Other Planner Options       |     8
Statistics / Cumulative Query and Index Statistics |     7
Query Tuning / Genetic Query Optimizer     |     7
Reporting and Logging / When to Log        |     7
Connections and Authentication / Authentication |     7
Version and Platform Compatibility / Previous PostgreSQL Versions |     7
Replication / Sending Servers              |     6
Write-Ahead Log / Checkpoints              |     6
Lock Management                             |     5
Statistics / Monitoring                     |     5
File Locations                              |     5
Connections and Authentication / TCP Settings |     5
Resource Usage / Cost-Based Vacuum Delay   |     5
Error Handling                              |     4
Client Connection Defaults / Shared Library Preloading |     4
Resource Usage / Background Writer         |     4
Write-Ahead Log / Archiving                |     4
Client Connection Defaults / Other Defaults |     3
Write-Ahead Log / Archive Recovery         |     3
Replication / Subscribers                  |     3
Write-Ahead Log / Recovery                 |     2
Reporting and Logging / Process Title      |     2
Version and Platform Compatibility / Other Platforms and Clients |     1
Resource Usage / Kernel Resources          |     1
Replication / Primary Server               |     1
Resource Usage / Disk                      |     1
(43 строки)
```

Для продолжения вывода (вместо промпта команда показывает двоеточие)

последовательно нажмите на клавиатуре клавиши `<z><q>`.

Категория **Customized Options** - это параметры расширений и приложений.

8) Сколько параметров установлено в файлах параметров конфигурации?

```
postgres=# select sourcefile, count(*) from pg_settings group by sourcefile;
          sourcefile                                     | count
-----+-----
/var/lib/postgresql/tantor-se-17-replica/data1/postgresql.conf |    14
/var/lib/postgresql/tantor-se-17-replica/data1/postgresql.auto.conf |     6
(3 строки)
```

В файлах параметров конфигурации установлено $14+6=20$ параметров.

9) В файле `postgresql.conf` большое количество параметров закомментированы и откомментированы. Комментарии - это короткая, качественная, удобная (под рукой) справка.

Какие параметры конфигурации были считаны из **основного файла** параметров при запуске экземпляра?

```
postgres=# select name, setting, sourceline from pg_settings where sourcefile like
'%1.conf' order by sourceline;
 name | setting | sourceline
-----+-----+-----
max_connections | 100 | 65
shared_buffers | 16384 | 131
dynamic_shared_memory_type | posix | 154
min_wal_size | 80 | 258
log_timezone | Europe/Moscow | 613
DateStyle | ISO, DMY | 727
TimeZone | Europe/Moscow | 729
lc_messages | ru_RU.UTF-8 | 743
lc_monetary | ru_RU.UTF-8 | 745
lc_numeric | ru_RU.UTF-8 | 746
lc_time | ru_RU.UTF-8 | 747
default_text_search_config | pg_catalog.russian | 753
shared_preload_libraries | pg_stat_statements,pg_store_plans,auto_expl... | 834
logging_collector | on | 835
(14 строк)
```

sourceline - номер строки от начала файла. Номер строки удобен для поиска параметра и его редактирования.

Ту же информацию можно посмотреть в представлении `pg_file_settings`.

10) Выполните команду:

```
postgres=# select name, setting, sourceline, applied from pg_file_settings where
sourcefile like '%1.conf';
 name | setting | sourceline | applied
-----+-----+-----+-----
max_connections | 100 | 65 | t
shared_buffers | 128MB | 131 | t
dynamic_shared_memory_type | posix | 154 | t
max_wal_size | 1GB | 257 | f
min_wal_size | 80MB | 258 | t
log_timezone | Europe/Moscow | 613 | t
datestyle | iso, dmy | 727 | t
timezone | Europe/Moscow | 729 | t
lc_messages | ru_RU.UTF-8 | 743 | t
lc_monetary | ru_RU.UTF-8 | 745 | t
lc_numeric | ru_RU.UTF-8 | 746 | t
lc_time | ru_RU.UTF-8 | 747 | t
default_text_search_config | pg_catalog.russian | 753 | t
listen_addresses | * | 833 | f
shared_preload_libraries | pg_stat_statements,pg_store_plans,auto_expl... | 834 | t
logging_collector | on | 835 | t
(15 rows)
```

По какой причине может быть расхождение в количестве строк в приведенном примере **14** и **15**?

В `pg_file_settings` присутствует параметр `max_wal_size`. У вас расхождения может не быть, или они могут быть в других параметрах. Параметр в примере установлен в файле `postgresql.auto.conf`.

11) Пример содержимого файлов:

```
postgres=# \! cat $PGDATA/postgresql.conf | grep max_wal_size
max_wal_size = 1GB
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_wal_size = '512MB'
max_slot_wal_keep_size = '1024MB'
```

В обоих файлах присутствует параметр `max_wal_size`. Представление `pg_file_settings` выводит все незакомментированные параметры из всех файлов. В столбце `applied` для 257-й строки стоит "f". Это означает, что данная строка переопределяется последующей строкой с тем же именем параметра (или строкой в файле `postgresql.auto.conf`, содержимое которого переопределяет значения из `postgresql.conf`). В примере строку 257 перекрыла строка 4 из файла `postgresql.auto.conf`. В запросах мы не выводили содержимое этого (`postgresql.auto.conf`) файла (предикат `sourcefile like '%1.conf'`).

12) В представлениях бывает, что имеется много столбцов, и при построчном выводе они не помещаются в терминале. Можно воспользоваться расширенным режимом вывода.

Выполните команды:

```
postgres=# select * from pg_settings where name = 'max_wal_size'\gx
-[ RECORD 1 ]-----+-----
name           | max_wal_size
setting        | 512
unit           | MB
category       | Write-Ahead Log / Checkpoints
short_desc     | Sets the WAL size that triggers a checkpoint.
extra_desc     |
context        | sighup
vartype        | integer
source         | configuration file
min_val        | 2
max_val        | 2147483647
enumvals       |
boot_val       | 1024
reset_val      | 512
sourcefile     | /var/lib/postgresql/tantor-se-17/data/postgresql.auto.conf
sourceline     | 4
pending_restart | f
```

В этом примере видны все детали параметра: категория, короткое описание, контекст. Значение параметра было применено из 4-й строки файла `postgresql.auto.conf`.

13) Пример вывода значений без предиката (фильтра):

```
postgres=# select name, setting, substring(sourcefile, 39) file, sourceline, applied from
pg_file_settings where name='max_wal_size';
-----+-----+-----+-----+-----
name      | setting | file                               | sourceline | applied
-----+-----+-----+-----+-----
max_wal_size | 1GB    | postgresql.conf                   | 257       | f
max_wal_size | 512MB  | postgresql.auto.conf              | 4         | t
(2 строки)
```

Представление `pg_file_settings` показывает **все** строки файлов параметров конфигурации, где устанавливаются значения параметров (незакомментированные и непустые строки). Для каждого параметра может быть несколько строк, в которых задаются значения этого параметра, это не является ошибкой, хотя этого стоит избегать (чтобы не было неоднозначности).

При запуске экземпляра (и перечитывании файлов, если значение параметра может быть изменено без рестарта экземпляра) применяется значение из файла `postgresql.auto.conf`, которое идёт самым последним. В этом файле также могут быть повторения, они появляются при редактировании файла вручную, а также в результате работы утилит (например, `pg_basebackup`), которые просто добавляют строки в конец файла, зная что установленное в конец файла будет превалировать.

Если в файле `postgresql.auto.conf` параметр отсутствует, то применяется значение, которое ближе к концу файла `postgresql.conf`.

Представление `pg_settings` показывает одну строку для каждого параметра, то есть ту, которая применена или может быть применена.

В столбце `pending_restart` значение `"t"` появится, если значение параметра было изменено в файле параметров конфигурации, файлы были перечитаны (**без перечитывания содержимое `pg_settings` не меняется**), и после перечитывания требуется перезапуск экземпляра (то есть у параметра `context=postmaster`). Во всех остальных случаях значение `pending_restart="f"`.

В отличие от `pg_settings` представление **`pg_file_settings` показывает текущее содержимое файлов параметров**, и в столбце `error` можно посмотреть, нет ли ошибок после редактирования файлов, что приведёт к невозможности запуска экземпляра.

14) Ошибок в этих двух файлах параметров конфигурации нет, если запрос типа `select sourcefile, sourceline, error from pg_file_settings where error is not null` не выдаст ни одной строки.

Если запрос выдаст одну строку, это не означает что ошибка только в одной строке, ошибок может быть много. После исправления ошибки нужно повторить запрос, добиваясь, чтобы запрос не выдал ни одной строки. Во многих случаях наличие ошибки приведёт к невозможности запуска экземпляра после его остановки.

Примеры (выполнять команды этого пункта не нужно):

Ошибка в значении параметра:

```
postgres=# \! cat $PGDATA/postgresql.auto.conf | grep max_wal
max_wal_size = '512mB'
```

```
postgres=# select substring(sourcefile, 39) file, sourceline, error from pg_file_settings
where error is not null;
```

file	sourceline	error
postgresql.auto.conf	4	setting could not be applied

(1 строка)

Не исправив предыдущую ошибку, добавлена ошибка в названии параметра:

```
postgres=# \! cat $PGDATA/postgresql.conf | grep 512MB
max_wol_size = 512MB

postgres=# select substring(sourcefile, 39) file, sourceline, error from pg_file_settings
where error is not null;
   file      | sourceline |          error
-----+-----+-----
 postgresql.conf |          836 | unrecognized configuration parameter
(1 строка)
```

Не исправив предыдущие ошибки, добавлена ошибка в синтаксисе строки:

```
postgres=# \! cat $PGDATA/postgresql.conf | grep max_wol
max_wol_size = 512MB
max_wol_size - 512MB

postgres=# select substring(sourcefile, 39) file, sourceline, error from pg_file_settings
where error is not null;
   file      | sourceline |          error
-----+-----+-----
 postgresql.conf |          837 | syntax error
(1 строка)
```

Если хоть одна из перечисленных ошибок будет присутствовать, то экземпляр не сможет запускаться после остановки или в процессе перезапуска:

```
postgres@tantor:~$ sudo systemctl restart tantor-se-server-17
Job for tantor-se-server-17.service failed because the control process exited with error
code.
See "systemctl status tantor-se-server-17.service" and "journalctl -xe" for details.
```

Ошибки "setting could not be applied" не всегда означают невозможность запуска экземпляра.

Часть 2. Параметры конфигурации с единицей измерения

1) Посмотрим, как менять значение параметров с единицей измерения.

Посмотрите свойства параметра `shared_buffers`:

```
postgres=# select * from pg_settings where name = 'shared_buffers' \gx
-[ RECORD 1 ]-----+-----
name                | shared_buffers
setting              | 16384
unit                 | 8kB
category             | Resource Usage / Memory
short_desc           | Sets the number of shared memory buffers used by the server.
extra_desc           |
context              | postmaster
vartype              | integer
source               | configuration file
min_val              | 16
max_val              | 1073741823
enumvals             |
boot_val             | 16384
reset_val            | 16384
sourcefile           | /var/lib/postgresql/tantor-se-17/data/postgresql.conf
sourceline           | 131
pending_restart     | f
```

Значение измеряется в блоках размером **8 КБ**. Параметр **целочисленный**.

2) Установите значение для этого параметра 12800:

```
postgres=# alter system set shared_buffers = 12800;
ALTER SYSTEM
```

3) Проверьте, что было записано в файл параметров:

```
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
...
shared_buffers = '12800'
```

Значение было введено без апострофов, в файле было установлено в **апострофах**.

4) Посмотрите, нет ли ошибок в установленном значении параметра:

```
postgres=# select substring(sourcefile, 39) file, sourceline, error from pg_file_settings
where error is not null;
-----+-----+-----
file                | sourceline | error
-----+-----+-----
postgresql.auto.conf |          6 | setting could not be applied
(1 строка)
```

Ошибка означает, что лучше использовать значения с единицами измерения, например `'100MB'`.

Экземпляр при этом успешно перезапустится и ошибка исчезнет.

4) Параметр имеет контекст `postmaster`, значит для изменения значения требует рестарт экземпляра. Перезапустите экземпляр:

```
postgres=# \q
postgres@tantor:~$ sudo systemctl restart tantor-se-server-17
[sudo] password for postgres: postgres
postgres@tantor:~$ psql
```

5) Посмотрите, какое значение у параметра после рестарта экземпляра:

```
postgres=# show shared_buffers;
shared_buffers
-----
100MB
(1 строка)
```

Значение выдаётся в мегабайтах. В файле параметров установлено значение '12800'.
 $12800 * 8192$ (8 КБ) = 104857600. $104857600 / 1024 / 1024 = 100$. 12800 блоков - это ровно 100 МБ.

6) Без единиц измерения этот параметр измеряется в блоках.

Установим значение в мегабайтах. Выполните команду:

```
postgres=# alter system set shared_buffers = 100mb;
ERROR: trailing junk after numeric literal at or near "100m"
СТРОКА 1: alter system set shared_buffers = 100mb;
      ^
```

Не получается. Для единиц измерения важен регистр. Выполните команду:

```
postgres=# alter system set shared_buffers = 100MB;
ERROR: trailing junk after numeric literal at or near "100M"
СТРОКА 1: alter system set shared_buffers = 100MB;
      ^
```

Не получается. Поставьте апострофы:

```
postgres=# alter system set shared_buffers = '100MB';
ALTER SYSTEM
```

Получилось.

Вы выполняли несколько раз команду, чтобы лучше запомнить особенность ввода значений параметров с единицами измерений: важен регистр единиц измерения и необходимы апострофы. Не забыв об этом, для этого параметра часто пытаются ввести число, интуитивно полагая, что значение задаётся в байтах (а оно в блоках) и получают нехватку памяти при рестарте экземпляра.

Между числом и единицей измерения могут быть пробелы и это не вызовет ошибок.

Например (выполнять не нужно):

```
postgres=# alter system set shared_buffers = '100      MB';
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_wal_size = '512MB'
max_slot_wal_keep_size = '1024MB'
shared_buffers = '100      MB'
```

Пробелы ухудшают читаемость.

7) Посмотрите, что было записано в файл при вводе значения с единицей измерения и в апострофах:

```
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
..
shared_buffers = '100MB'
```

8) Уберите из `postgresql.auto.conf` параметр `shared_buffers`:

```
postgres=# alter system reset shared_buffers;
ALTER SYSTEM
```

```
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
```

Строки (если их было несколько, что может получиться при редактировании файла вручную) с параметром `shared_buffers` исчезнут.

В этом пункте мы изучили, как убирать параметры из `postgresql.auto.conf`.

Часть 3. Параметры конфигурации логического типа

1) Посмотрим параметр **логического типа (bool)**:

```
postgres=# select * from pg_settings where name = 'hot_standby_feedback'\gx
-[ RECORD 1 ]-----+-----
name           | hot_standby_feedback
setting        | off
unit           |
category       | Replication / Standby Servers
short_desc     | Allows feedback from a hot standby to the primary that will avoid query
conflicts      |
extra_desc     |
context        | sighup
vartype        | bool
source         | default
min_val        |
max_val        |
enumvals       |
boot_val       | off
reset_val      | off
sourcefile     |
sourceline     |
pending_restart | f
```

Контекст `sighup` означает, что для применения нового значения достаточно перечитать файлы конфигурации.

2) "Включим" параметр, то есть установим значение в `true`:

```
postgres=# alter system set hot_standby_feedback = on;
ERROR:  parameter "hot_standby_feedback" requires a Boolean value
```

Ошибка означает, что нельзя сократить значение, так как есть неоднозначность. В качестве значения допускаются `on` и `off`:

```
postgres=# alter system set hot_standby_feedback = on;
ALTER SYSTEM
```

Значение `on` допустимо для логических параметров. Проверьте, что допустимы и другие значения:

```
postgres=# alter system set hot_standby_feedback = 1;
ALTER SYSTEM
postgres=# alter system set hot_standby_feedback = '1';
ALTER SYSTEM
```

Единица тоже допустима:

```
postgres=# alter system set hot_standby_feedback = tr;
ALTER SYSTEM
```

Допускаются сокращения значений, но только если нет неоднозначности.

Неоднозначность была с сокращением до одной буквы "o".

3) Посмотрите, что было записано в файл параметров:

```
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
..
hot_standby_feedback = 'tr'
```

Значение в сокращенном виде было записано в апострофах.

Сокращения неудобны для чтения. Для логических параметров лучше использовать канонические значения `on`, `off`.

4) Перечитайте файлы параметров, чтобы новое значение начало действовать:

```
postgres=# select pg_reload_conf();
pg_reload_conf
-----
t
(1 строка)
```

```
postgres=# show hot_standby_feedback;
hot_standby_feedback
-----
on
(1 строка)
```

Значение установилось правильно.

Часть 4. Конфигурационные параметры

"Параметры конфигурации" (settings) и "Конфигурационные параметры" (config) созвучны. В этой части практики рассмотрим "Конфигурационные параметры".

Есть три способа посмотреть конфигурационные параметры: утилита командной строки `pg_config`, представление `pg_config` и функция `pg_config()`.

1) Посмотрите, какие параметры конфигурации существуют, утилитой `pg_config`:

```
postgres@tantor:~$ pg_config --help
```

`pg_config` предоставляет информацию об установленной версии PostgreSQL.

Использование:

```
pg_config [ПАРАМЕТР]...
```

Параметры:

<code>--bindir</code>	показать расположение исполняемых файлов
<code>--docdir</code>	показать расположение файлов документации
<code>--htmldir</code>	показать расположение HTML-файлов документации
<code>--includedir</code>	показать расположение файлов-заголовков (.h) для клиентских интерфейсов на языке C
<code>--pkgincludedir</code>	показать расположение других файлов-заголовков (.h)
<code>--includedir-server</code>	показать расположение файлов-заголовков (.h) для сервера
<code>--libdir</code>	показать расположение библиотек объектного кода
<code>--pkglibdir</code>	показать расположение динамически загружаемых модулей
<code>--localedir</code>	показать расположение файлов описания локалей
<code>--mandir</code>	показать расположение справочных страниц
<code>--sharedir</code>	показать расположение платформенно-независимых файлов
<code>--sysconfdir</code>	показать расположение общесистемных файлов конфигурации
<code>--pgxs</code>	показать расположение makefile для расширений
<code>--configure</code>	показать параметры скрипта "configure", с которыми был собран PostgreSQL
<code>--cc</code>	показать, с каким значением CC собран PostgreSQL
<code>--cppflags</code>	показать, с каким значением CPPFLAGS собран PostgreSQL
<code>--cflags</code>	показать, с какими флагами C собран PostgreSQL
<code>--cflags_sl</code>	показать, с каким значением CFLAGS_SL собран PostgreSQL
<code>--ldflags</code>	показать, с каким значением LDFLAGS собран PostgreSQL
<code>--ldflags_ex</code>	показать, с каким значением LDFLAGS_EX собран PostgreSQL
<code>--ldflags_sl</code>	показать, с каким значением LDFLAGS_SL собран PostgreSQL
<code>--libs</code>	показать, с каким значением LIBS собран PostgreSQL
<code>--version</code>	показать версию PostgreSQL
<code>-, --help</code>	показать эту справку и выйти

При запуске без аргументов выводятся все известные значения.

Эти параметры устанавливаются при сборке Tantor Postgres и не меняются. Они одинаковы для сборок `BE`, `SE`, `SE1C`. Поскольку названия директорий длинные и запомнить их сложно, то польза от утилиты `pg_config` в том, что можно, зная название утилиты и название вида директории, получить путь в файловой системе к нужной директории.

2) Запустите утилиту без параметров, утилита выдаст значения всех параметров:

```
postgres@tantor:~$ pg_config
```

```
BINDIR = /opt/tantor/db/17/bin
```

```
DOCDIR = /opt/tantor/db/17/share/doc/postgresql
```

```
HTMLDIR = /opt/tantor/db/17/share/doc/postgresql
```

```
INCLUDEDIR = /opt/tantor/db/17/include
```

```
PKGINCLUDEDIR = /opt/tantor/db/17/include/postgresql
```

```
INCLUDEDIR-SERVER = /opt/tantor/db/17/include/postgresql/server
```

```
LIBDIR = /opt/tantor/db/17/lib
```

```
PKGLIBDIR = /opt/tantor/db/17/lib/postgresql
```

```
LOCALEDIR = /opt/tantor/db/17/share/locale
```

```
MANDIR = /opt/tantor/db/17/share/man
```

```

SHAREDIR = /opt/tantor/db/17/share/postgresql
SYSCONFDIR = /opt/tantor/db/17/etc/postgresql
PGXS = /opt/tantor/db/17/lib/postgresql/pgxs/src/makefiles/pgxs.mk
CONFIGURE = '--prefix=/opt/tantor/db/16' '--enable-tap-tests' '--enable-nls=en ru' '--
with-python' '--with-icu' '--with-lz4' '--with-zstd' '--with-ssl=openssl' '--with-ldap'
'--with-pam' '--with-uuid=e2fs' '--with-libxml' '--with-libxslt' '--with-gssapi' '--with-
selinux' '--with-systemd' '--with-llvm' 'CFLAGS=-O2 -pipe -Wno-missing-braces'
'LLVM_CONFIG=/usr/bin/llvm-config-11' 'CLANG=/usr/bin/clang-11' 'PYTHON=/usr/bin/python3'
CC = gcc
CPPFLAGS = -D_GNU_SOURCE -I/usr/include/libxml2
CFLAGS = -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -
Werror=vla -Wendif-labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-
function-type -Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -
fexcess-precision=standard -Wno-format-truncation -Wno-stringop-truncation -O2 -pipe -
Wno-missing-braces
CFLAGS_SL = -fPIC
LDFLAGS = -L/usr/lib/llvm-11/lib -Wl,--as-needed -Wl,-rpath,'/opt/tantor/db/17/lib',--
enable-new-dtags
LDFLAGS_EX =
LDFLAGS_SL =
LIBS = -lpqcommon -lpqport -lselinux -lzstd -llz4 -lxml2 -lpam -lssl -lcrypto -
lgssapi_krb5 -lz -lreadline -lpthread -lrt -ldl -lm
VERSION = PostgreSQL 17.5
    
```

Местоположение директории с внешними библиотеками (подгружаемыми модулями, **PKGLIBDIR**) показывает параметр **--pkglibdir**.

3) Библиотеки загружаются при запуске экземпляра параметром конфигурации **shared_preload_libraries** или, если библиотека может загружаться не только при запуске экземпляра, но и динамически серверным процессом, то командой **LOAD 'имя_библиотеки'**;

Посмотрите, какие библиотеки доступны:

```

postgres@tantor:~$ ls $(pg_config --pkglibdir)
adminpack.so          latin_and_mic.so      pg_visibility.so
amcheck.so           libpqwalreceiver.so  pg_wait_sampling.so
auth_delay.so        llvmljit.so           pg_walinspect.so
auto_explain.so      llvmljit_types.bc    pgxml.so
autoinc.so           lo.so                 pgxs
basebackup_to_shell.so ltree_plpython3.so   plpgsql.so
basic_archive.so     ltree.so              plpython3.so
bitcode              moddatetime.so        postgres_fdw.so
bloom.so             old_snapshot.so       refint.so
btree_gin.so         orafce.so             seg.so
btree_gist.so        pageinspect.so        sepgsql.so
citext.so            page_repair.so        sslinfo.so
credcheck.so         passwordcheck.so      tablefunc.so
cube.so              pgauditlogtofile.so  tcn.so
cyrillic_and_mic.so  pgaudit.so           test_decoding.so
dblink.so            pg_background.so      tsm_system_rows.so
dict_int.so          pg_buffercache.so     tsm_system_time.so
dict_snowball.so     pg_columnar.so        unaccent.so
dict_xsyn.so         pg_cron.so            utf8_and_big5.so
earthdistance.so     pgcrypto.so           utf8_and_cyrillic.so
euc2004_sjis2004.so  pg_freespacemap.so   utf8_and_euc2004.so
euc_cn_and_mic.so    pg_hint_plan.so      utf8_and_euc_cn.so
euc_jp_and_sjis.so   pgoutput.so           utf8_and_euc_jp.so
euc_kr_and_mic.so    pg_partman_bgw.so     utf8_and_euc_kr.so
euc_tw_and_big5.so   pg_prewarm.so         utf8_and_euc_tw.so
file_fdw.so          pgq_lowlevel.so       utf8_and_gb18030.so
fuzzystrmatch.so     pgq_triggers.so       utf8_and_gbk.so
hstore_plpython3.so  pg_qualstats.so       utf8_and_iso8859_1.so
hstore.so            pg_repack.so          utf8_and_iso8859.so
http.so              pgrowlocks.so         utf8_and_johab.so
hypopg.so            pg_stat_statements.so utf8_and_sjis2004.so
insert_username.so   pgstattuple.so        utf8_and_sjis.so
_int.so              pg_store_plans.so     utf8_and_uhc.so
    
```

```
isn.so                pg_surgery.so        utf8_and_win.so
jsonb_plpython3.so   pg_trgm.so           uuid-osp.so
latin2_and_win1250.so pg_variables.so
```

4) Проверим, что некоторые разделяемые библиотеки можно загружать динамически.

Загрузите модуль `pg_hint_plan`:

```
postgres=# show pg_hint_plan.enable_hint;
ERROR: unrecognized configuration parameter "pg_hint_plan.enable_hint"
```

Серверный процесс не знает о таком параметре, поскольку модуль не был загружен ни серверным процессом, ни на уровне экземпляра. Загрузите модуль в память серверного процесса, обслуживающего текущую сессию:

```
postgres=# LOAD 'pg_hint_plan';
LOAD
```

Библиотека загрузилась в память серверного процесса, обслуживающего сессию, в которой была дана команда. В этой сессии можно пользоваться функционалом модуля.

5) В частности, теперь в сессии доступны параметры конфигурации модуля. При наборе текста можно использовать клавишу табуляции на клавиатуре `<TAB>`, `psql` продолжит за вас набор текста, если других вариаций нет, а при двойном нажатии клавиши покажет список возможных значений.

Наберите `show pg_hint<TAB>.<TAB>`:

```
postgres=# show pg_hint_plan.enable_hint;
 pg_hint_plan.enable_hint
-----
 on
(1 строка)
```

6) Воспользуемся и другим вариантом просмотра параметров конфигурации:

```
postgres=# \dconfig pg_hint_plan.*

Список параметров конфигурации
Параметр                | Значение
-----+-----
pg_hint_plan.debug_print | off
pg_hint_plan.enable_hint | on
pg_hint_plan.enable_hint_table | off
pg_hint_plan.hints_anywhere | off
pg_hint_plan.message_level | log
pg_hint_plan.parse_messages | info
(6 строк)
```

При инсталляции расширений в директорию `PGGLIBDIR` скопированы динамически линкуемые библиотеки (`*.so`), если расширение содержит разделяемые библиотеки.

Вторая полезная директория при администрировании расширений - это `SHAREDIR`. В эту директорию копируются файлы расширений, которые затем устанавливаются командой `CREATE EXTENSION`.

7) Расширения не являются общим объектом кластера и устанавливаются на уровне баз данных.

Посмотрите, какие расширения готовы к установке в базы данных:

```
postgres@tantor:~$ ls $(pg_config --sharedir)/extension | grep .control
adminpack.control
amcheck.control
...
xml2.control
```

8) Список тех же самых расширений можно посмотреть в представлении

```
pg_available_extensions:
postgres=# select count(*) from pg_available_extensions;
count
-----
      69
```

9) Посмотрите определение представления:

```
postgres=# \sv pg_available_extensions
CREATE OR REPLACE VIEW pg_catalog.pg_available_extensions AS
SELECT e.name,
       e.default_version,
       x.extversion AS installed_version,
       e.comment
FROM   pg_available_extensions() e(name, default_version, comment)
LEFT JOIN pg_extension x ON e.name = x.extname
```

Представление использует функцию `pg_available_extensions()`, которая читает содержимое файлов `*.control` в директории `SHAREDIR`.

9) Посмотрите определение функции:

```
postgres=# \sf pg_available_extensions()
CREATE OR REPLACE FUNCTION pg_catalog.pg_available_extensions(OUT name name, OUT
default_version text, OUT comment text)
RETURNS SETOF record
LANGUAGE internal
STABLE PARALLEL SAFE STRICT COST 10 ROWS 100
AS $function$pg_available_extensions$function$
```

Командой `\sv` - вы можете смотреть тексты представлений.

Командой `\sf` - тексты подпрограмм, в том числе системного каталога.

Часть 5. Файл служб

Если возникают затруднения с копированием файла из-за привилегий на уровне операционной системы или с редактированием файлов, то эту часть практики можно не выполнять, а посмотреть нижеприведенные примеры.

1) Посмотрите, на какую директорию указывает параметр `sysconfdir`. В этой директории находятся файлы по умолчанию.

```
postgres@tantor:~$ pg_config --sysconfdir
/opt/tantor/db/17/etc/postgresql
```

2) Создайте директорию:

```
postgres@tantor:~$ sudo mkdir /opt/tantor/db/17/etc
postgres@tantor:~$ sudo chown postgres:postgres /opt/tantor/db/17/etc
postgres@tantor:~$ mkdir /opt/tantor/db/17/etc/postgresql
```

3) Скопируйте файл примера в эту директорию (команда одной строкой):

```
postgres@tantor:~$ cp $(pg_config --sharedir)/pg_service.conf.sample $(pg_config --sysconfdir)/pg_service.conf
```

4) Посмотрите содержимое файла служб:

```
postgres@tantor:~$ cat $(pg_config --sysconfdir)/pg_service.conf
#
#      Connection configuration file
#
# A service is a set of named connection parameters.  You may specify
# multiple services in this file.  Each starts with a service name in
# brackets.  Subsequent lines have connection configuration parameters of
# the pattern "param=value" or LDAP URLs starting with "ldap://"
# to look up such parameters.  A sample configuration for postgres is
# included in this file.  Lines beginning with '#' are comments.
#
# Copy this to your sysconf directory (typically /usr/local/pgsql/etc) and
# rename it pg_service.conf.
#
#[postgres]
#dbname=postgres
#user=postgres
```

5) Отредактируйте файл `/opt/tantor/db/17/etc/postgresql/pg_service.conf`:

```
postgres@tantor:~$ mcedit /opt/tantor/db/17/etc/postgresql/pg_service.conf
```

6) Вставьте в файл строки:

```
[postgres]
dbname=postgres
user=postgres
host=/var/run/postgresql
port=5432
```

Теперь имеется определение службы с названием "postgres". Можно указать несколько служб в этом файле. В параметре `host` можно указывать IP-адрес или имя хоста. При указании **директории** используется локальное соединение через Unix-сокеты.

7) Воспользуйтесь именем службы для соединения с базой данных. Выполните команду:

```
postgres@tantor:~$ psql service=postgres
psql (17.5)
Введите "help", чтобы получить справку.
postgres=# \conninfo
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5432"
```

Если в файле служб допустить ошибку, например, указать порт как **5435**, то выдастся ошибка:

```
postgres@tantor:~$ psql service=postgres
psql: ошибка: подключиться к серверу через сокет
"/var/run/postgresql/.s.PGSQL.5435" не удалось: Нет такого файла или каталога
```

Сервер действительно работает локально и принимает подключения через этот сокет?

8) Файл служб также может находиться в домашнем каталоге пользователя операционной системы (`~/.pg_service.conf`). Точка в начале названия файла нужна.

Директория `SYSCONFDIR` также используется для файла с именем `psqlrc`. При запуске **без параметра -x** утилита `psql`, после подключения к базе данных, читает и выполняет команды из `psqlrc`, а затем из файла `~/.psqlrc` (если эти файлы есть). Этими файлами можно воспользоваться для настройки свойств сессии `psql`.

Глава 4а. Логическая структура кластера

Часть 1. Установка параметров конфигурации на различных уровнях

Цель этой части изучить, как устанавливать параметры конфигурации на различных уровнях, и какие уровни превалируют.

1) Установите промпт, который будет показывать пользователя и базу данных, с которой создана сессия (текст после `\set` вводится одной строкой):

```
postgres=# \set PROMPT1 '% [%033[0;31m%] %n% [%033[0m%] @% [%033[0;36m%] %/ [%033[0m%]
% [%033[0;33m%] % [%033[5m%] %x% [%033[0m%] % [%033[0m%] %R%# '
postgres=# \set PROMPT2 '% [%033[0;31m%] %n% [%033[0m%] @% [%033[0;36m%] %/ [%033[0m%]
% [%033[0;33m%] % [%033[5m%] %x% [%033[0m%] % [%033[0m%] %R%# '

```

2) Добавьте в конец файла `postgresql.conf` параметр:

```
postgres=# \! echo "my.level = 'Pgconf'" >> $PGDATA/postgresql.conf

```

Обязательно проверьте, что используете две угловые скобки `>>`, а не одну - иначе затрёте содержимое файла.

Параметр `my.level` - это "параметр приложения", имя которому мы сами придумали. В имени обязательно должна быть точка, иначе экземпляр не запустится.

Если не добавить параметр в `postgresql.conf` и не перечитать файл параметров, то команда `ALTER SYSTEM SET my.level = 'строка'`; выдаст ошибку:

```
ERROR: unrecognized configuration parameter "my.level"

```

Такая ошибка выдаётся, если с момента запуска экземпляра не была загружена разделяемая библиотека, которая бы зарегистрировала бы параметры конфигурации. Загрузка осуществляется параметром `shared_preload_libraries` или командой `LOAD`.

3) Проверьте, что строка добавилась:

```
postgres=# \! tail -n 1 $PGDATA/postgresql.conf
my.level = 'Pgconf'

```

5) Перечитайте файлы параметров:

```
postgres=# select pg_reload_conf();
pg_reload_conf

```

```
-----

```

```
t
(1 строка)

```

6) Посмотрите, какие типы (context) параметров есть:

```
postgres=# select distinct context from pg_settings;
context

```

```
-----

```

```
postmaster
superuser-backend
user
internal
backend
sighup
superuser

```

(7 строк)

Большинство параметров типа "user" можно устанавливать на всех уровнях. Однако, могут быть нюансы. Например, параметр `application_name` устанавливает клиентское приложение после создания сессии. Для утилиты `psql` это значение `psql`. Поэтому устанавливать значение этого параметра на уровне кластера, базы, роли, роли в базе бессмысленно, так как установка на уровне сессии перекрывает эти значения. Его можно устанавливать на уровне сессии, транзакции, функции.

Параметр `temp_tablespaces` можно устанавливать на любом уровне, но он имеет особенность: при создании подпрограммы на языке `plpgsql` (у этого языка есть функция- "обёртка", проверяющая тело подпрограммы в момент создания) проверяется наличие табличных пространств, и если их нет, то подпрограмма не создаётся.

Параметры типа `internal` не меняются.

Параметры типа `postmaster` меняются с перезапуском экземпляра и их можно менять командой `ALTER SYSTEM`.

Параметры типа `sighup` меняются командой `ALTER SYSTEM`, но требуют перечитывания файлов параметров.

7) Создайте объекты следующими командами:

```
drop database IF EXISTS bob;
drop ROLE IF EXISTS bob;
drop database IF EXISTS rob;
drop user IF EXISTS rob;
CREATE USER bob SUPERUSER LOGIN;
CREATE ROLE rob SUPERUSER LOGIN;
CREATE DATABASE bob OWNER bob STRATEGY WAL_LOG;
CREATE DATABASE rob OWNER rob STRATEGY FILE_COPY;
\c bob bob
CREATE SCHEMA IF NOT EXISTS bob AUTHORIZATION bob;
CREATE SCHEMA IF NOT EXISTS rob AUTHORIZATION rob;
\dconfig my.level
alter system set my.level = 'System';
select pg_reload_conf();
alter database bob set my.level = 'Database';
alter role bob set my.level = 'Role';
alter role bob in database bob set my.level = 'RoleInDatabase';
CREATE OR REPLACE FUNCTION bob.bob()
RETURNS text
LANGUAGE plpgsql
SET my.level TO 'Function'
AS $function$
BEGIN
    RAISE NOTICE 'my.level %', current_setting('my.level');
    RAISE NOTICE 'search_path %', current_schemas(true);
    RETURN current_setting('my.level');
END;
$function$
;

CREATE OR REPLACE FUNCTION bob.bobdef()
RETURNS text
LANGUAGE plpgsql
SECURITY DEFINER
AS $function$
BEGIN
    RAISE NOTICE 'my.level %', current_setting('my.level');
    RAISE NOTICE 'search_path %', current_schemas(true);
```

```

RAISE NOTICE 'current_user %', current_user;
RAISE NOTICE 'session_user %', session_user;
RAISE NOTICE 'user %', user;
RETURN current_setting('my.level');
END;
$function$
;
    
```

С помощью этих объектов будем проверять, с какого уровня будут забираться параметры конфигурации.

Уровень функции перекрывает все уровни.

Следующий уровень, который перекрывает остальные (кроме функции) - это уровень транзакции `SET LOCAL`.

Следующий уровень - сессии. Если вызывать функции `SECURITY DEFINER`, которые работают с правами владельца, то уровень сессии вызывающего перекроет значения сессии владельца.

А если на сессии не устанавливать значение, то чьё значение будет действовать - роли-владельца (`DEFINER`)?

Нет, будет действовать значение параметра, установившееся на уровне сессии того, кто вызывает функцию. Если параметр был установлен на "роль в базе", то он и установится в сессии. Если не был установлен, то установится "на роль". Дальше "на базу". Это важно знать. Для функций и процедур особенно важно значение параметра `search_path`, которое будет действовать в теле функции или процедуры. Функции и процедуры в постгрес называются подпрограммами.

Вторая проблема. Значение по умолчанию у `search_path="$user", public`.

Значение `$user` в теле подпрограммы у `SECURITY DEFINER` - имя роли-владельца.

Поэтому со значением `$user` путь поиска у подпрограмм с `DEFINER` и `INVOKER` различны. При этом, вызывающий подпрограмму может установить в своей сессии `search_path` без `$user`. Путь поиска станет другим в теле подпрограммы.

Поэтому **с `SECURITY DEFINER` подпрограммами лучше не полагаться на путь поиска, а всегда устанавливать путь поиска в определении подпрограммы**. Можно было бы использовать префикс с названием схем перед каждым объектом в теле подпрограммы, но тогда нужно ставить префикс и в теле всех подпрограмм, которые она вызывает, в том числе подпрограммы системного каталога. Иначе вызывающий может установить `search_path = myschema, public, pg_catalog` и заменить любую подпрограмму системного каталога на свою в схеме `myschema`. Также вызывающий может создать временную таблицу и она перекроет любые таблицы, поэтому при создании `SECURITY DEFINER` подпрограммы нужно не забывать о `pg_temp` и в определении подпрограммы **всегда указывать его явно и последним**, например: `search_path = pg_catalog, схема_владельца, pg_temp`.

Текст кажется сложным для понимания? Архитектурные уязвимости часто непонятны архитекторам программных систем, иначе бы они их не допускали. В вышеприведенном примере создания функции `bobdef()` с правами создателя заложена уязвимость. Перед вызовом `bobdef()`

можно создать функцию `схема.current_setting(text)`. Перед вызовом `bobdef` дать команду `set search_path=схема, public, pg_catalog` и `bobdef()` вызовет созданную функцию с правами владельца `bobdef`.

8) Посмотрите значения, которые были установлены вышеприведенным набором команд:

```
postgres=# \drds
          Список параметров
 Роль | БД |          Параметры
-----+-----+-----
 bob  | bob | my.level=RoleInDatabase
 bob  |     | my.level=Role
      | bob | my.level=Database
(3 строки)
```

Если вы планируете обращать внимание на безопасность или менять параметры на разных уровнях, то стоит запомнить команду `\drds`.

9) Изменения начнут действовать только при создании новой сессии. Переподсоединяясь, посмотрим параметры какого уровня действуют в сессии. Подсоединимся под пользователем `rob` к базе `bob`:

```
bob@bob=# \c bob rob
```

Вы подключены к базе данных "`bob`" как пользователь "`rob`".

10) Функция `bob()` в схеме `bob` была создана с установкой параметра в значение **Function**. Независимо от того, как вызывается функция и независимо от того `INVOKER` она или `DEFINER`, в её теле будет действовать то, что установлено в её определении:

```
rob@bob=# SELECT bob.bob() as "my.level";
NOTICE: my.level Function
NOTICE: search_path {pg_catalog, rob, public}
 my.level
-----
 Function
(1 строка)
```

Путь поиска в теле функции вызывающего её пользователя (`rob`), ведь функция типа `INVOKER`.

11) Вызовем функцию `DEFINER`:

```
rob@bob=# SELECT bob.bobdef() as "my.level";
NOTICE: my.level Database
NOTICE: search_path {pg_catalog, bob, public}
NOTICE: current_user bob
NOTICE: session_user rob
NOTICE: user bob
 my.level
-----
 Database
(1 строка)
```

Подумайте, почему уровень `Database`?

Задаваться вопросами полезно, так как это активирует память. Мы изучаем хоть и простые правила, но у них много комбинаций. Схожие утверждения запоминаются плохо, и просто читать задание и выполнять команды, не задумываясь, неинтересно.

12) Для ответа на вопрос можно проверить, какое значение установлено в текущей сессии:

```
rob@bob=# SHOW my.level;
```

```
my.level
```

```
-----
Database
(1 строка)
```

Установлен уровень **Database**, поэтому и в теле функции применяется значение с этого уровня.

Мы ответили на предыдущий вопрос, но возник новый. Почему параметр взят с уровня базы?

Потому что мы не устанавливали значения параметра для пользователя `rob` (в пункте 7 можно посмотреть команды, которыми делались настройки) ни на уровне роли, ни на уровне роли в базе. Мы это делали для пользователя `bob`.

Но также мы устанавливали параметр на уровне базы. Уровень базы перекрывает уровень кластера (значение `"System"`).

13) Поменяем в текущей сессии значение и повторим вызов функции:

```
rob@bob=# SET my.level = 'Session';
SET
rob@bob=# SELECT bob.bobdef() as "my.level";
NOTICE: my.level Session
NOTICE: search_path {pg_catalog,bob,public}
NOTICE: current_user bob
NOTICE: session_user rob
NOTICE: user bob
my.level
-----
Session
(1 строка)
```

Функция использует параметр, действующий в сессии.

Путь поиска у `DEFINER` функции - её владельца, из-за `search_path = '$user, public'` установленного по умолчанию на уровне кластера.

Функция `current_user` тоже выдаёт для `DEFINER` владельца функции. А `session_user` - вызывающего. При написании кода функции она может получить имя вызывающей её роли и использовать это знание.

14) Проверим функцию `bob.bob()`:

```
rob@bob=# SELECT bob.bob() as "my.level";
NOTICE: my.level Function
NOTICE: search_path {pg_catalog,rob,public}
my.level
-----
Function
(1 строка)
```

Для неё ничего не поменялось, она неизменно использует уровень `Function`.

15) Вдруг вызов этой функции поменял значение на `Function` на уровне сессии и не вернул его обратно? Проверим:

```
rob@bob=# SHOW my.level;
my.level
-----
Session
(1 строка)
```

То, что в теле функции параметр имел другое значение, не повлияло на сессию.

16) Проверим функцией `current_setting`:

```
rob@bob=# SELECT current_setting('my.level');
```

```
current_setting
```

```
-----
Session
(1 строка)
```

Результат тот же.

17) Проверим, не повлияет ли установка параметра на уровне транзакции на параметр, установленный на уровне функции:

```
rob@bob =# BEGIN TRANSACTION;
BEGIN
rob@bob *=# SET LOCAL my.level = 'Transaction';
SET
rob@bob *=# SELECT bob.bob() as "my.level";
NOTICE: my.level Function
NOTICE: search_path {pg_catalog,rob,public}
my.level
-----
Function
(1 строка)
```

Не повлияет. Параметр, установленный на уровне функции, превалирует.

18) Для функций, где нет установки на их уровне, он будет действовать:

```
rob@bob *=# SELECT bob.bobdef() as "my.level";
NOTICE: my.level Transaction
NOTICE: search_path {pg_catalog,bob,public}
NOTICE: current_user bob
NOTICE: session_user rob
NOTICE: user bob
my.level
-----
Transaction
(1 строка)
```

19) Завершим транзакцию и проверим значение параметра:

```
rob@bob *=# END;
COMMIT
rob@bob =# SHOW my.level;
my.level
-----
Session
(1 строка)
```

Значение вернулось в `session`, то есть то значение, которое было до изменения на уровне транзакции (`SET LOCAL`).

20) Подсоединимся пользователем `bob` к базе `postgres`. На уровне этой базы мы параметр не меняли. Откуда будет браться значение?

```
rob@bob =# \c postgres bob
Вы подключены к базе данных "postgres" как пользователь "bob".

bob@postgres =# SHOW my.level;
my.level
-----
Role
(1 строка)
```

Значение берется из установленного для роли `bob`.

Почему? Для базы данных `postgres` параметр не устанавливался.

21) Уберем установку параметра для роли `bob`:

```
bob@postgres=# ALTER ROLE bob RESET my.level;
ALTER ROLE
```

Если переподсоединиться, то параметр будет взят с уровня кластера, значение - System.

Не будем это проверять.

22) Подсоединимся к базе bob. Откуда будет взят параметр?

```
bob@postgres=# \c bob bob
You are now connected to database "bob" as user "bob".
bob@bob=# SHOW my.level;
my.level
-----
RoleInDatabase
(1 строка)
```

Параметр установлен и на базу и на роль в базе. Более детальный превагирует.

23) Подсоединимся к базе rob:

```
bob@bob=# \c rob bob
You are now connected to database "rob" as user "bob".
bob@rob=# SHOW my.level;
my.level
-----
System
(1 строка)
```

На базе rob установок нет, а для пользователя bob мы чуть раньше (пункт 21) [убрали](#) [установку со значением "Role"](#).

24) Уберем установку на роль в базе:

```
bob@rob=# ALTER ROLE bob IN DATABASE bob RESET my.level;
ALTER ROLE
bob@rob=# SHOW my.level;
my.level
-----
System
(1 строка)
```

В этой базе и без убирания было бы то же самое.

25) А в базе bob? Проверим:

```
bob@rob=# \c bob bob
You are now connected to database "bob" as user "bob".
bob@bob=# SHOW my.level;
my.level
-----
Database
(1 строка)
```

После убирания параметра на уровне "роль в базе" стал действовать уровень базы.

26) Уберём его и на уровне базы и проверим:

```
bob@bob=# ALTER DATABASE bob RESET my.level;
ALTER DATABASE
bob@bob=# SHOW my.level;
my.level
-----
Database
(1 строка)
```

Осталось прежнее значение, так как забыли переподсоединиться.

27) Переподсоединяемся:

```
bob@bob =# \c bob bob
You are now connected to database "bob" as user "bob".
bob@bob =# SHOW my.level;
my.level
-----
System
(1 строка)
```

Теперь берётся с уровня кластера.

28) Уберём параметр из файла postgresql.auto.conf:

```
bob@bob =# alter system reset my.level;
ALTER SYSTEM
bob@bob =# select pg_reload_conf();
pg_reload_conf
-----
t
(1 строка)
```

Но у нас параметр установлен в postgresql.conf и оттуда мы его не убирали.

29) Проверим, что в случае отката транзакции, откатывается команда установки параметра на уровне сессии:

```
bob@bob =# begin;
BEGIN
bob@bob *=# set my.level='forRollback';
SET
bob@bob *=# show my.level;
my.level
-----
forRollback
(1 строка)

bob@bob *=# rollback;
ROLLBACK
bob@bob =# show my.level;
my.level
-----
Pgconf
(1 строка)

bob@bob =# end;
WARNING: there is no transaction in progress
COMMIT
```

Команда `end` эквивалентна команде `commit`, но редко используется.

30) Можно задаться вопросом: что с установками на уровне кластера?

Ответ: команда установки параметра на уровне кластера не работает в транзакции, поэтому откатываться не может. Проверим:

```
bob@bob =# begin;
BEGIN
bob@bob *=# alter system set my.level = 'forRollback';
ERROR: ALTER SYSTEM cannot run inside a transaction block
bob@bob !=# end;
ROLLBACK
```

Почему на команду `end` серверный процесс вернул сообщение `ROLLBACK`? Если бы вместо `end` была бы дана команда `commit`, сообщение было бы тоже `ROLLBACK`, так как транзакция перешла в состояние сбоя, на что указывает символ `"!"`.

31) Удалите созданные объекты, выполнив команды:

```
\c bob postgres
drop schema rob;
\c postgres postgres
drop database if exists rob;
drop database if exists bob;
drop user if exists bob;
drop database if exists rob;
drop user if exists rob;
```

Часть 2. Установка пути поиска в функциях и процедурах

1) Выполните команды:

```
CREATE USER rob LOGIN;
CREATE OR REPLACE FUNCTION bobdef()
  RETURNS text
  LANGUAGE plpgsql
  SECURITY DEFINER
AS $function$
BEGIN
  RAISE NOTICE 'search_path %', current_schemas(true);
  RAISE NOTICE 'current_user %', current_user;
  RAISE NOTICE 'session_user %', session_user;
  RAISE NOTICE 'user %', user;
  RETURN now();
END;
$function$
;
grant create on schema public to rob;
```

Команды создают непривилегированного пользователя `rob` с правом подсоединяться к базам и дают ему право создавать объекты в схеме `public` базы `postgres`.

2) Подсоединитесь пользователем `rob` к базе `postgres` и проверьте, что функция `bobdef()` выполняется как запрограммировано при ее создании:

```
postgres=# \c postgres rob
Вы подключены к базе данных "postgres" как пользователь "rob".
postgres=> SELECT bobdef();
NOTICE: search_path {pg_catalog,public}
NOTICE: current_user postgres
NOTICE: session_user rob
NOTICE: user postgres
          bobdef
-----
...44.401115+03
(1 строка)
```

3) Создайте под непривилегированным пользователем `rob` следующую функцию:

```
postgres=>
CREATE OR REPLACE FUNCTION public.now() RETURNS text
LANGUAGE plpgsql
AS $$
BEGIN
  RAISE NOTICE 'now() user %', user;
  ALTER USER ROB SUPERUSER;
  RETURN 'done';
END;
$$;
```

4) Поменяйте путь поиска, вызовите функцию `bobdef()`. Функция вызовет созданную пользователем `rob` функцию `now()`, которая выполнится с правами владельца функции

`bobdef()`, то есть с правами пользователя `postgres`:

```
postgres=> set search_path = public, pg_catalog;
SET
postgres=> SELECT bobdef();
NOTICE: search_path {public,pg_catalog}
NOTICE: current_user postgres
NOTICE: session_user rob
NOTICE: user postgres
NOTICE: now() user postgres
          bobdef
-----
done
```

(1 строка)

5) Проверьте атрибуты пользователя `rob` после вызова функции:

```
postgres=> \du rob
          Список ролей
Имя роли | Атрибуты
-----+-----
rob      | Суперпользователь
```

Чтобы подпрограмма `SECURITY DEFINER` была безопасна, `search_path` должен:

- 1) быть установлен на уровне определения (а не после `BEGIN`) подпрограммы;
- 2) исключать любые схемы, доступные для создания или изменения пользователями с

меньшим уровнем привилегий, чем у владельца такой подпрограммы;

3) схема `pg_temp` должна быть указана явно в конце пути поиска, установленного в определении подпрограммы.

Пример установки параметра на уровне подпрограммы:

```
\c postgres postgres
CREATE OR REPLACE FUNCTION bobdef()
  RETURNS text
  LANGUAGE plpgsql
  SECURITY DEFINER
  SET search_path = pg_catalog, pg_temp
AS $function$
BEGIN
  RAISE NOTICE 'search_path %', current_schemas(true);
  RAISE NOTICE 'current_user %', current_user;
  RAISE NOTICE 'session_user %', session_user;
  RAISE NOTICE 'user %', user;
  RETURN now();
END;
$function$
;
```

Эта подпрограмма безопасна.

6) Удалите созданные объекты:

```
\c postgres postgres
drop function if exists public.now();
revoke create on schema public from rob;
drop user rob;
```

Глава 4b. Физическая структура кластера

Часть 1. Создание соединения с базой данных

1) Настройте параметры хранения WAL-сегментов.

```
postgres=# alter system set max_slot_wal_keep_size = '128MB';
ALTER SYSTEM
postgres=# alter system set max_wal_size = '128MB';
ALTER SYSTEM
postgres=# ALTER SYSTEM SET idle_in_transaction_session_timeout = '100min';
ALTER SYSTEM
postgres=# select pg_reload_conf();
 pg_reload_conf
-----
 t
(1 строка)
postgres=# select pg_switch_wal();
 pg_switch_wal
-----
 7/941FBFF2
(1 строка)
```

Эти настройки нужны, чтобы в процессе работы с большими объемами данных, не возникло нехватки места в директории журналов PGDATA/pg_wal.

2) Посмотрите, по какому сетевому адресу осуществляется прослушивание:

```
postgres=# \dconfig list*
List of configuration parameters
Parameter | Value
-----+-----
listen_addresses | localhost
(1 строка)
```

Прослушивание осуществляется по локальному сетевому интерфейсу.

3) Посмотрите, на каком порту осуществляется прослушивание:

```
postgres=# \dconfig port
List of configuration parameters
Parameter | Value
-----+-----
port      | 5432
(1 строка)
```

Порт 5432 по умолчанию.

4) Посмотрите, по какому адресу мы подсоединились:

```
postgres=# \conninfo
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5432".
```

Мы подсоединились через **Unix-сокет**.

5) Посмотрите, как выглядит файл, создаваемый процессом postgres:

```
postgres=# \! ls -al /var/run/postgresql
total 4
drwxrwsr-x  2 postgres postgres  80 .
drwxr-xr-x 29 root      root    800 ..
srwxrwxrwx  1 postgres postgres   0 .s.PGSQL.5432
-rw-----  1 postgres postgres  80 .s.PGSQL.5432.lock
```

Создаётся два файла, их нельзя удалить.

6) Местоположение файлов определяется параметром конфигурации

`unix_socket_directories`. Посмотрите значение этого параметра:

```
postgres=# \dconfig unix_socket*
      Список параметров конфигурации
      Параметр                | Значение
-----+-----
 unix_socket_directories | /var/run/postgresql
 unix_socket_group      |
 unix_socket_permissions    | 0777
(3 строки)
```

С помощью этих параметров можно настроить возможность использовать локальное подключение пользователям операционной системы. По умолчанию **0777** разрешает подключаться **любым пользователям операционной системы, в которой запущен экземпляр**. По умолчанию **имя группы** пусто, и группой для файла сокета является основная группа пользователя, запускающего экземпляр: `postgres`.

Полное описание параметров есть в документации:

https://docs.tantorlabs.ru/tdb/ru/17_5/se/runtime-config-connection.html#RUNTIME-CONFIG-CONNECTION-SETTINGS

7) Если сообщения `psql` выдаются на английском языке, то в окне терминала операционной системы () установите вывод сообщений утилит на русском языке, чтобы удобнее было читать справочную информацию по параметрам утилит:

```
postgres=# \q
postgres@tantor:~$ locale -a | grep ru
ru_RU.utf8
postgres@tantor:~$ export LC_MESSAGES=ru_RU.utf8
```

8) Посмотрите, с какими параметрами можно создать базу данных:

```
postgres@tantor:~$ createdb --help
createdb создаёт базу данных PostgreSQL.
```

Использование:

```
createdb [ПАРАМЕТР]... [ИМЯ_БД] [ОПИСАНИЕ]
```

Параметры:

```
-D, --tablespace=ТАБЛ_ПРОСТР табличное пространство по умолчанию для базы данных
-e, --echo                    отображать команды, отправляемые серверу
-E, --encoding=КОДИРОВКА     кодировка базы данных
-l, --locale=ЛОКАЛЬ           локаль для базы данных
--lc-collate=ЛОКАЛЬ          параметр LC_COLLATE для базы данных
--lc-ctype=ЛОКАЛЬ            параметр LC_CTYPE для базы данных
--icu-locale=ЛОКАЛЬ          локаль ICU для базы данных
--icu-rules=ПРАВИЛА          настройка правил сортировки ICU
--locale-provider={libc|icu} провайдер локали для основного правила сортировки БД
-O, --owner=ВЛАДЕЛЕЦ         пользователь-владелец новой базы данных
-S, --strategy=STRATEGY      стратегия создания базы данных: wal_log или file_copy
-T, --template=ШАБЛОН        исходная база данных для копирования
-V, --version                 показать версию и выйти
-?, --help                   показать эту справку и выйти
```

Параметры подключения:

```
-h, --host=ИМЯ                имя сервера баз данных или каталог сокетов
-p, --port=ПОРТ              порт сервера баз данных
-U, --username=ИМЯ           имя пользователя для подключения к серверу
-w, --no-password            не запрашивать пароль
-W, --password                запросить пароль
--maintenance-db=ИМЯ_БД     сменить опорную базу данных
```

По умолчанию именем базы данных считается имя текущего пользователя.

-t задает имя базы, клон которой хочется получить.

-s позволяет существенно уменьшить объем журналов, если шаблон или клонируемая база данных большого размера.

--maintenance-db к какой из баз кластера нужно подключиться утилите, чтобы дать команду `CREATE DATABASE`.

Часть 2. Содержимое табличного пространства

1) Создайте директорию:

```
postgres=# \! mkdir $PGDATA/../u01
```

Проверьте, что пользователь `postgres` может **читать-писать** в эту директорию:

```
postgres=# \! ls -al $PGDATA/../u01
итого 8
drwxr-xr-x 2 postgres postgres 4096 .
drwxr-xr-x 6 postgres postgres 4096 ..
```

2) Создайте табличное пространство:

```
postgres=# CREATE TABLESPACE u01tbs LOCATION '/var/lib/postgresql/tantor-se-17/u01';
CREATE TABLESPACE
```

3) Посмотрите содержимое директории табличного пространства:

```
postgres=# \! ls -al $PGDATA/../u01
итого 12
drwx----- 3 postgres postgres 4096 .
drwxr-xr-x 6 postgres postgres 4096 ..
drwx----- 2 postgres postgres 4096 PG_17_642505061
```

Была создана поддиректория с названием `pg_17_642505061`. В имени поддиректории присутствует **номер основной версии** `postgres`. Такие директории и создаются и удаляются автоматически, чтобы упростить обновление программного обеспечения на новую основную версию.

4) Создайте в табличном пространстве таблицу:

```
postgres=# drop table if exists t;
NOTICE: table "t" does not exist, skipping
DROP TABLE
postgres=# CREATE TABLE t (id bigserial, t text) TABLESPACE u01tbs;
CREATE TABLE
```

5) Наполните таблицу данными:

```
postgres=# INSERT INTO t(t) SELECT encode((floor(random()*1000)::numeric ^
100)::numeric)::text::bytea, 'base64') from generate_series(1,5000000);

INSERT 0 5000000
```

Было вставлено 5 млн. строк.

6) Посмотрим, какие файлы появились. Откройте второй терминал, переключитесь в пользователя `postgres`, и перейдите в директорию табличного пространства и базы данных:

```
postgres@tantor:~$ cd $PGDATA/../u01/Pg_17_642505061/5
postgres@tantor:~/tantor-se-17/u01/Pg_17_642505061/5$ ls -al
```

```
total 1952072
drwxr-x--- 2 postgres postgres      4096 12:02 .
drwxr-x--- 3 postgres postgres      4096 11:47 ..
-rw-r----- 1 postgres postgres 1073741824 12:03 365769
-rw-r----- 1 postgres postgres  924581888 12:04 365769.1
-rw-r----- 1 postgres postgres  507904 12:02 365769_fsm
-rw-r----- 1 postgres postgres   65536 12:04 365769_vm
-rw-r----- 1 postgres postgres      0 12:01 365773
-rw-r----- 1 postgres postgres   8192 12:01 365774
```

Файл с суффиксом ".1". Это второй файл основного слоя (main fork).

7) Вставьте ещё миллион строк:

```
postgres=# INSERT INTO t(t) SELECT encode((floor(random()*1000)::numeric ^
100::numeric)::text::bytea, 'base64') from generate_series(1,1000000);
INSERT 0 1000000
```

8) Посмотрите, какие файлы появились в директории табличного пространства:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ ls -al
total 2342372
drwxr-x--- 2 postgres postgres      4096 12:06 .
drwxr-x--- 3 postgres postgres      4096 11:47 ..
-rw-r----- 1 postgres postgres 1073741824 12:05 365769
-rw-r----- 1 postgres postgres 1073741824 12:06 365769.1
-rw-r----- 1 postgres postgres  250404864 12:06 365769.2
-rw-r----- 1 postgres postgres   606208 12:06 365769_fsm
-rw-r----- 1 postgres postgres   65536 12:06 365769_vm
-rw-r----- 1 postgres postgres      0 12:01 365773
-rw-r----- 1 postgres postgres   8192 12:01 365774
```

Добавился файл с суффиксом ".2". Это третий файл основного слоя.

9) Посмотрите утилитой `oid2name` информацию о файле:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ oid2name -f 365769
From database "postgres":
  Filenode  Table Name
-----
  365769    t
```

Это полезно в случае, когда вы видите файл в файловой системе, находящийся в директории табличного пространства и хотите узнать, к какому объекту какой базы данных относится файл. Например, вы видите большое количество файлов по 2 ГБ основного слоя и предполагаете, что какой-то объект неоправданно разросся (bloat), и хотите найти этот объект.

Также это полезно когда вы хотите удалить табличное пространство, а оно не удаляется, потому что в нём в каких-то базах данных имеются объекты. Список объектов команда удаления не выдаст:

```
postgres=# drop tablespace u01tbs;
ERROR:  tablespace "u01tbs" is not empty
```

Список баз данных, в которых есть объекты, можно определить по именам поддиректорий в директории табличного пространства, в которых имеются файлы. Названия поддиректорий - это `oid` баз данных.

10) Посмотрите утилитой `oid2name` информацию о таблице:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ oid2name -t t
From database "postgres":
  Filenode  Table Name
-----
  365769           t
```

Это полезно, если вы хотите найти названия файлов основного слоя таблицы.

11) В директории имеются ещё файлы.

Та же типичная задача: есть файл в директории, хочется узнать к какому объекту файл относится.

Посмотрите, что выдаёт утилита про оставшиеся файлы:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ oid2name -f 365773
From database "postgres":
  Filenode  Table Name
-----
  365773  pg_toast_365769
```

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ oid2name -f 365774
From database "postgres":
  Filenode  Table Name
-----
  365774  pg_toast_365769_index
```

Это файлы **TOAST-таблицы** и **TOAST-индекса**. Для таблицы (обычного типа heap) может быть создана одна TOAST-таблица и один индекс на эту TOAST таблицу.

20) В директории присутствуют файлы слоёв **vm** и **fsm**:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ ls
365769 365769.1 365769.2 365769_fsm 365769_vm 365773 365774
```

12) Посмотрим, можно ли удалить эти файлы.

Остановите экземпляр:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ pg_ctl stop
waiting for server to shut down.... done
server stopped
```

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ rm *_*
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ ls
365769 365769.1 365769.2 365773 365774
```

Файлы `_vm` и `_fsm` удалены.

13) Запустите экземпляр:

```
postgres@tantor:~/tantor-se-17/u01/PG_17_642505061/5$ sudo systemctl start
tantor-se-server-17.service
[sudo] password for postgres: postgres
```

После запуска экземпляра файлы не появились.

14) Во втором окне, где запущен `psql`, переподсоединимся и обратимся к таблице:

```
postgres=# select count(*) from t;
 count
-----
 6000000
(1 строка)
```

Команда полностью просмотрела страницы файлов основного слоя и не выдала ошибку.

Файлы `_vm` и `_fsm` опять не появились. Может они не нужны и без них всё хорошо работает?

15) Выполните вакуумирование таблицы:

```
postgres=# vacuum verbose analyze t;
```

```
INFO: vacuuming "postgres.public.t"
INFO: finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 292711 remain, 292711 scanned (100.00% of total)
tuples: 0 removed, 6000001 remain, 0 are dead but not yet removable, oldest xmin: 2117
removable cutoff: 2117, which was 0 XIDs old when operation ended
new relminmxid: 250029, which is 732 MXIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 367.632 MB/s, avg write rate: 367.745 MB/s
buffer usage: 292841 hits, 292617 misses, 292707 dirtied
WAL usage: 292712 records, 10 full page images, 19106735 bytes
system usage: CPU: user: 3.40 s, system: 2.04 s, elapsed: 6.21 s
INFO: vacuuming "postgres.pg_toast.pg_toast_365769"
INFO: finished vacuuming "postgres.pg_toast.pg_toast_365769": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 2117
removable cutoff: 2117, which was 0 XIDs old when operation ended
new relfrozenxid: 2117, which is 41 XIDs ahead of previous value
new relminmxid: 250029, which is 732 MXIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
avg read rate: 12.480 MB/s, avg write rate: 0.000 MB/s
buffer usage: 19 hits, 1 misses, 0 dirtied
WAL usage: 1 records, 0 full page images, 202 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: analyzing "public.t"
INFO: "t": scanned 30000 of 292711 pages, containing 614828 live rows and 0 dead rows; 30000 rows in sample,
5998897 estimated total rows
VACUUM
```

Файлы `_vm` и `_fsm` появились.

Файлы этих слоёв могут отсутствовать сразу после создания объекта. Файл `fsm` может быть создан серверным процессом, который использует этот файл для поиска блока со свободным местом для вставки строк. Файлы могут создаваться в любой момент, как только процесс автовакуума начнет обрабатывать объект. Процесс автовакуума начнет обрабатывать объект после вставки или изменения и удаления какого-то (устанавливается параметрами конфигурации и параметрами на уровне таблицы) количества строк в этом объекте.

Удалять файлы `vm` и `fsm` вручную не нужно, такой потребности нет.

Доступ к блокам файлов объектов постоянного хранения **всех слоёв** производится через буферный кэш в разделяемой области памяти, поэтому перед удалением файлов мы останавливали экземпляр.

Часть 3. Файл объекта "последовательность"

При создании таблицы тип первого столбца был указан как `bigserial`. Это означает, что значение столбца заполняется последовательностью.

1) Посмотрите определение таблицы:

```
postgres=# \d t
```

Таблица "public.t"				
Столбец	Тип	Правило сортировки	Допустимость NULL	По умолчанию
id	bigint		not null	nextval('t_id_seq'::regclass)
t	text			

Табличное пространство: "u01tbs"

2) Посмотрите определение последовательности:

```
postgres=# \ds+
```

Список отношений						
Схема	Имя	Тип	Владелец	Хранение	Размер	Описание
public	t_id_seq	последовательность	postgres	постоянное	8192 bytes	

(1 строка)

У последовательности есть размер, а значит физически она представляет собой файл **размером один блок**.

3) Посмотрите характеристики последовательности как "объекта" (**отношения**, класса):

```
postgres=# select * from pg_class where relname='t_id_seq' \gx
```

- [RECORD 1] -	
oid	374239
relname	t_id_seq
relnamespace	2200
reltype	0
reloftype	0
relowner	10
relam	0
relfilenode	374239
reltablespace	0
relpages	1
reltuples	1
relallvisible	0
reltoastrelid	0
relhasindex	f
relisshared	f
relpersistence	p
relkind	S
relnatts	3
relchecks	0
relhasrules	f
relhastriggers	f
relhassubclass	f
relrowsecurity	f
relforcerowsecurity	f
relispopulated	t
relreplident	n
relispartition	f
relrewrite	0

Мы получили `oid`, **номер файла**, `oid` табличного пространства (**ноль** означает табличное пространство по умолчанию для базы данных). Также видно, что последовательность физически представляет собой **одну запись** (**reltuples**) в **одном блоке** (**relpages**).

4) Посмотрите путь к файлу последовательности:

```
postgres=# SELECT pg_relation_filepath(374239);
```

```
pg_relation_filepath
-----
base/5/374239
(1 строка)
```

5) То же самое можно получить и не обращаясь к `pg_class` за `oid` последовательности.

Для этого можно использовать приведение типов:

```
postgres=# SELECT pg_relation_filepath('t_id_seq'::text::regclass);
pg_relation_filepath
-----
base/5/374239
(1 строка)
```

Файл последовательности был создан и располагается в табличном пространстве `pg_default`, которое является табличным пространством по умолчанию для базы данных

postgres:

```
postgres=# select dattablespace, datname from pg_database;
dattablespace | datname
-----+-----
1663 | postgres
1663 | test_db
1663 | template1
1663 | template0
(4 строки)
```

```
postgres=# select oid, spcname from pg_tablespace;
oid | spcname
-----+-----
1663 | pg_default
1664 | pg_global
18651 | u01tbs
(3 строки)
```

Часть 4. Перемещение таблицы в другое табличное пространство

Переместим таблицу `t` в табличное пространство `pg_default`.

В окне терминала будем проверять сколько места занимает кластер.

1) В окне терминала перейдите в директорию `/var/lib/postgresql/tantor-se-17`:

```
postgres@tantor:~$ cd $PGDATA/..
postgres@tantor:~/tantor-se-17$ du -hs
3.2G
```

В этом окне будем нажимать на клавиатуре стрелку вверх и клавишу `<ENTER>`, пока работает команда перемещения.

2) В окне `psql` в целях оценки, сколько журнальных данных сгенерируется, посмотрим текущий LSN:

```
postgres=# SELECT pg_current_wal_lsn();
pg_current_wal_lsn
-----
4/E2BFA2A0
(1 строка)
```

3) В окне `psql` дайте команду перемещения. Воспользуйтесь, например, синтаксисом перемещения всех таблиц:

```
postgres=# alter table ALL IN TABLESPACE u01tbs SET TABLESPACE pg_default;
```

4) Пока команда работает, переключитесь в окно терминала и выполните команду `watch -n1 du -hs`, чтобы посмотреть сколько места занимает кластер в процессе переноса файлов таблицы:

```
postgres@tantor:~/tantor-se-17$ watch -n1 du -hs
4.1G
4.4G
4.6G
4.9G
5.1G
5.4G
3.2G
```

Для возврата в терминал наберите `ctrl+c`.

Занятое кластером место увеличилось как минимум на 2.2Gb, с 3.2G до 5.4G.

Если вы не успели выполнить команды, можно посмотреть приведённые числа. Если интересно попробовать самостоятельно, то можно повторять команды

```
alter table t SET TABLESPACE u01tbs;
alter table t SET TABLESPACE pg_default;
```

перемещая файлы таблицы повторно из одного табличного пространства в другое.

На время перемещения размер кластера увеличился как минимум на размер перемещаемой таблицы. Размеры файлов сегментов журнала были ограничены в начале практики, иначе они бы ещё больше увеличивали занятое место в процессе выполнения команды перемещения.

5) Посмотрите текущий LSN:

```
postgres=# SELECT pg_current_wal_lsn();
pg_current_wal_lsn
-----
5/731A7860
(1 строка)
```

6) Посчитайте, какой объем данных прошел через журналы:

```
postgres=# select pg_size_pretty('5/731A7860'::pg_lsn - '4/E2BFA2A0'::pg_lsn);
pg_size_pretty
-----
2310 MB
(1 строка)
```

7) Посмотрите размер таблицы:

```
postgres=# select pg_size_pretty(pg_total_relation_size('t'));
pg_size_pretty
-----
2287 MB
(1 строка)
```

Через журнал кластера прошел весь объем перемещаемых данных. Если бы параметром `max_wal_size` не было установлено ограничение на максимальный размер журналов в начале практики, то использовалось бы дополнительно место "двойного размера" перемещаемых данных (4.5Gb), так же как при применении утилиты `pg_repack`.

Часть 5. Перемещение таблицы в другое табличное пространство утилитой `pg_repack`

1) Установите расширение:

```
postgres=# create extension pg_repack;
CREATE EXTENSION
```

2) Запустите утилиту:

```
postgres@tantor:~$ pg_repack -t t
WARNING: relation "public.t" must have a primary key or not-null unique keys
```

3) Утилита не может работать с таблицами без первичного ключа. Добавьте первичный

ключ:

```
postgres=# ALTER TABLE t ADD CONSTRAINT t_pk PRIMARY KEY (id);
ALTER TABLE
```

Добавление первичного ключа создало уникальный индекс.

4) Верните утилитой таблицу в табличное пространство `u01tbs`:

```
postgres@tantor:~$ pg_repack -t t -s u01tbs
INFO: repacking table "public.t"
```

Объем места, которое было занято на время работы (2.3G), не изменится по сравнению с перемещением командой `ALTER TABLE` - в пике занято примерно 5.6Гб с 3.3Гб.

Индекс на таблицу при этом не был перемещён, так как мы использовали параметр `"-t"`.

5) Посмотрите как работает параметр `"-I"`:

```
postgres@tantor:~$ pg_repack -I t -s u01tbs
INFO: repacking table "public.t"
```

Объем места увеличивался до 5.7Гб.

6) Файлов в табличном пространстве стало больше:

```
postgres@tantor:~$ ls $PGDATA/./u01/PG_17_642505061/5
374064 374067 374068 374085 374085.1 374085.2 374085_fsm 374088 374089
```

Файл слоя `vm` отсутствует, так как не было вакуумирования.

7) Выполните анализ (сбор статистики для оптимизатора) таблицы `t`:

```
postgres=# analyze t;
ANALYZE
```

Количество файлов не поменялось.

Анализ не создал файл слоя `vm`.

7) Выполните вакуумирование таблицы `t`:

```
postgres=# vacuum t;
VACUUM
```

Добавился файл `374085_vm`.

Часть 6. Использование утилиты pgcompacttable

Предварительная настройка.

1) Дайте разрешения на выполнение утилиты:

```
postgres@tantor:~$ sudo chmod 755 -R /opt/tantor/db/17/tools/pgcompacttable
```

2) Установите требуемое для работы утилиты стандартное расширение:

```
postgres=# create extension pgstattuple;
CREATE EXTENSION
```

3) Проверьте, что утилита запускается:

```
postgres@tantor:~$ /opt/tantor/db/17/tools/pgcompacttable/bin/pgcompacttable --help
```

Name:

```
pgcompacttable - PostgreSQL bloat reducing tool.
```

Usage:

```
pgcompacttable [OPTION...]
```

General options:

```
[-?mV] [(-q | -v LEVEL)]
```

Connection options:

```
[-h HOST] [-p PORT] [-U USER] [-W PASSWD] [-P PATH]
```

Targeting options:

```
(-a | -d DBNAME...) [-n SCHEMA...] [-t TABLE...] [-N SCHEMA...] [-T
TABLE...]
```

Examples:

```
Shows usage manual.
```

```
pgcompacttable --man
```

Compacts all the bloated tables in all the database in the cluster plus their bloated indexes. Prints additional progress information.

```
pgcompacttable --all --verbose info
```

Compacts all the bloated tables in the billing database and their bloated indexes except ones that are in the pgq schema.

```
pgcompacttable --dbname billing --exclude-schema pgq
```

4) Если утилита не запускается, то установите библиотеки, которые она использует для работы командой:

```
postgres@tantor:~$ sudo apt-get install libdbi-perl libdbd-pg-perl
```

Чтение списков пакетов Готово

Построение дерева зависимостей

Чтение информации о состоянии Готово

Уже установлен пакет libdbd-pg-perl самой новой версии (3.7.4-3).

Уже установлен пакет libdbi-perl самой новой версии (1.642-1+deb10u2).

Обновлено 0 пакетов, установлено 0 новых пакетов, для удаления отмечено 0 пакетов, и 2 пакетов не обновлено.

5) Внесите изменения в таблицу:

```
postgres=# update t set id = id+6000000;
```

```
UPDATE 6000000
```

```
postgres=# delete from t where id < 11000000;
```

```
DELETE 4999999
```

6) Получите размер таблицы и её индексов:

```
postgres=# select pg_size_pretty(pg_total_relation_size('t'));
```

```
pg_size_pretty
```

```
-----
```

```
4881 MB
```

```
(1 строка)
```

7) Посмотрите список файлов таблицы:

```
postgres=# \! ls -l --color -w 1 $PGDATA/../../u01/Pg_17_642505061/5
```

```

итого 4671504
-rw----- 1 postgres postgres 1073741824 12:13 18797
-rw----- 1 postgres postgres 1073741824 12:13 18797.1
-rw----- 1 postgres postgres 1073741824 12:13 18797.2
-rw----- 1 postgres postgres 1073741824 12:13 18797.3
-rw----- 1 postgres postgres 487276544 12:11 18797.4
-rw----- 1 postgres postgres 1196032 12:10 18797_fsm
-rw----- 1 postgres postgres 147456 12:11 18797_vm
-rw----- 1 postgres postgres 0 11:51 18800
-rw----- 1 postgres postgres 8192 11:51 18801
    
```

Количество файлов и их общий размер увеличились.

Если запустить утилиту, она может долго работать. Поскольку утилита предназначена для использования с минимальным влиянием на работу экземпляра, можно будет, пока работает утилита, в параллельной сессии посмотреть, какие блокировки она устанавливает и продолжать следующие пункты практики. Если ожидание затянется, можно будет рестартовать экземпляр и усечь таблицу командой `TRUNCATE`.

8) Запустите утилиту командой с числом циклов 1 (по умолчанию 10):

```

postgres@tantor:~$ /opt/tantor/db/17/tools/pgcompacttable/bin/pgcompacttable -T t -o 1 -E
0
[12:17:56] (postgres) Connecting to database
[12:17:57] (postgres) Postgres backend pid: 15709
[12:17:57] (postgres) Handling tables. Attempt 1
[12:17:57] (postgres:public.demo2) SQL Error: ERROR: only heap AM is supported
[12:17:57] (postgres:public.demo2) Table handling interrupt.
[12:17:57] (postgres:columnar_internal.chunk) Statistics: 22 pages (48 pages including toasts and
indexes)
[12:17:57] (postgres:columnar_internal.chunk) Reindex: columnar_internal.chunk_pkey, initial size
18 pages(144.000KB), has been reduced by 61% (88.000KB), duration 0 seconds.
[12:17:57] (postgres:columnar_internal.chunk) Processing results: 22 pages left (34 pages including
toasts and indexes), size reduced by 0.000B (112.000KB including toasts and indexes) in total.
[12:17:58] (postgres:public.hypo) Statistics: 55 pages (90 pages including toasts and indexes)
[12:17:58] (postgres:public.perf_columnar) SQL Error: ERROR: only heap AM is supported
[12:17:58] (postgres:public.perf_columnar) Table handling interrupt.
[12:17:58] (postgres:public.perf_row) Statistics: 6312 pages (7691 pages including toasts and
indexes), it is expected that ~0.570% (35 pages) can be compacted with the estimated space saving
being 286.746KB.
[12:18:09] (postgres:public.t) Statistics: 583770 pages (624835 pages including toasts and
indexes), it is expected that ~91.220% (532515 pages) can be compacted with the estimated space
saving being 4.063GB.
[12:19:09] (postgres:public.t) Progress: 14%, 75560 pages completed.
[12:20:09] (postgres:public.t) Progress: 31%, 165855 pages completed.
[12:21:09] (postgres:public.t) Progress: 53%, 282255 pages completed.
[12:22:09] (postgres:public.t) Progress: 64%, 341475 pages completed.
[12:23:09] (postgres:public.t) Progress: 82%, 437160 pages completed.
[12:23:59] (postgres:public.t) Reindex: public.t_pk, initial size 40888 pages(319.438MB), has been
reduced by 93% (297.992MB), duration 0 seconds.
[12:23:59] (postgres:public.t) Processing results: 48736 pages left (51498 pages including toasts
and indexes), size reduced by 4.082GB (4.374GB including toasts and indexes) in total.
[12:23:59] (postgres) Processing complete.
[12:23:59] (postgres) Processing results: size reduced by 4.082GB (4.374GB including toasts and
indexes) in total.
[12:23:59] (postgres) Disconnecting from database
[12:23:59] Processing complete: 1 retries to process has been done
[12:23:59] Processing results: size reduced by 4.082GB (4.374GB including toasts and indexes) in
total, 4.082GB (4.374GB) postgres.
    
```

Утилита работала дольше, чем перемещение таблицы - 6 минут, и освободила **4.374GB** в обоих табличных пространствах (таблица, индекс, TOAST, TOAST-индекс).

9) В другом окне терминала (если успеть), можно посмотреть, какие блокировки установлены:

```
postgres=# select locktype, database, relation, mode, granted from pg_locks;
```

locktype	database	relation	mode	granted
relation	5	12073	AccessShareLock	t
virtualxid			ExclusiveLock	t
relation	5	18761	RowExclusiveLock	t
relation	5	18706	AccessShareLock	t
relation	5	18706	RowExclusiveLock	t
relation	5	12104	AccessShareLock	t
virtualxid			ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t
advisory	5		ExclusiveLock	t

(19 строк)

```
postgres=# select relname, oid from pg_class where oid in (12073,18761,18706,12104) ;
```

relname	oid
t	18706
pg_settings	12104
pg_locks	12073

(3 строки)

На таблицу установлена блокировка самого щадящего уровня `ACCESS SHARE`. Такую блокировку устанавливает команда `SELECT`. Остальные блокировки служебные, и на работу с таблицей не влияют. Блокировку на свой виртуальный номер (`virtualxid`) всегда устанавливает любая транзакция. Рекомендательные блокировки (`advisory`) используются самой утилитой, чтобы исключить её параллельный запуск.

10) Можно также проверить, меняется ли объем занятого кластером места. В процессе работы утилиты место, занятое кластером, почти не увеличивалось, наоборот, оно может постепенно освобождаться. Это одно из основных преимуществ утилиты.

```
postgres@tantor:~/tantor-se-17$ du -hs
5.6G
```

После завершения работы утилиты место освободилось:

Проверим место:

```
postgres@tantor:~/tantor-se-17$ du -hs
1.3G
```

Место освободилось.

11) Распределение нагрузки на центральный процессор разумное (75% и 20%), использование языка `perl` не является узким местом:

```
postgres@tantor:~$ top
```

Для вывода нагрузки по процессорам - нажать на клавиатуре клавишу единица `<1>`.

Для выхода - нажать клавишу с буквой `<q>`.

```
top - 17:25:44 up 1 day, 6:51, 3 users, load average: 0.94, 1.11, 0.77
Tasks: 174 total, 2 running, 172 sleeping, 0 stopped, 0 zombie
%Cpu(s): 42.3 us, 4.4 sy, 0.0 ni, 53.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3913.6 total, 1674.0 free, 554.5 used, 1685.1 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used, 2919.2 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 20030 postgres 20   0 220640 153896 149532 R   75.1   3.8   4:28.88 postgres
 20029 postgres 20   0 35944  20556  7616 S   18.9   0.5   0:47.06 pgcompacttable
  1139 astra    20   0 44528  18468  4024 S    0.3   0.5   0:22.86 fly-wm
```

12) Удалите таблицу:

```
postgres=# drop table t;
DROP TABLE
```

Часть 7. Расширение pg_columnar (колоночно-ориентированный формат)

Часть 7а. Установка и использование pg_columnar

1) Установите расширение pg_columnar:

```
postgres=# create extension pg_columnar;
CREATE EXTENSION
```

Расширение добавляет табличный метод доступа columnar:

```
postgres=# SELECT * FROM pg_am WHERE amtype = 't';
 oid | amname | amhandler | amtype
-----+-----+-----+-----
    2 | heap   | heap_tableam_handler | t
 18276 | columnar | columnar_internal.columnar_handler | t
(2 строки)
```

2) В документации приводится пример функции на языке Python, которая генерирует данные. Установите поддержку языка:

```
postgres=# create extension plpython3u;
CREATE EXTENSION
```

3) Создайте функцию как в документации.

Текст функции и команд создания таблиц приведён в документации:

https://docs.tantorlabs.ru/tdb/ru/17_5/se/hydra.html

```
CREATE OR REPLACE FUNCTION random_words(n INT4) RETURNS TEXT LANGUAGE plpython3u AS $$
import random
t = ''
words = ['ноль', 'один', 'два', 'три', 'четыре', 'пять', 'шесть', 'семь', 'восемь',
'девять', 'десять']
for i in range(0,n):
    if (i != 0):
        t += ' '
    r = random.randint(0,len(words)-1)
    t += words[r]
return t
$$;
```

4) Создайте обычную таблицу, которая будет использоваться для сравнения:

```
CREATE TABLE perf_row(
  id INT8,
  ts TIMESTAMPTZ,
  customer_id INT8,
  vendor_id INT8,
  name TEXT,
  description TEXT,
```

```
value NUMERIC,
quantity INT4
) WITH (fillfactor = 100);
```

5) Создайте таблицу с колоночным способом хранения:

```
CREATE TABLE perf_columnar(LIKE perf_row) USING COLUMNAR;
```

6) Используя функцию, заполните таблицу данными:

```
INSERT INTO perf_row
SELECT
g, -- id
'2024-01-01'::timestampz + ('1 minute'::interval * g), -- ts
(random() * 1000000)::INT4, -- customer_id
(random() * 100)::INT4, -- vendor_id
random_words(5), -- name
random_words(30), -- description
(random() * 100000)::INT4/100.0, -- value
(random() * 100)::INT4 -- quantity
FROM generate_series(1,400000) g;
```

При выбранных значениях среднее количество строк на одной странице 18:

```
postgres=# select (ctid::text::point)[0]::int block, count((ctid::text::point)[1]::int)
from perf_row group by block limit 1;
 block | count
-----+-----
  1552 |    18
(1 строка)
```

6) Скопируйте данные в таблицу с колоночным форматом хранения:

```
INSERT INTO perf_columnar SELECT * FROM perf_row;
```

7) Сравните размер, занимаемый двумя таблицами:

```
postgres=# SELECT pg_total_relation_size('perf_row')::numeric /
pg_total_relation_size('perf_columnar');
?column?
-----
 6.6730711498048634
(1 строка)
```

Размер, занимаемый таблицей в колоночном формате, меньше в **6.6** раз.

8) Команда вакуумирования показывает **степень сжатия** данных:

```
postgres=# VACUUM VERBOSE perf_columnar;
postgres=# VACUUM VERBOSE perf_columnar;
INFO:  statistics for "perf_columnar":
storage id: 10000000004
total file size: 27303936, total data size: 27191296
compression rate: 6.14x
total row count: 400000, stripe count: 3, average rows per stripe: 133333
chunk count: 320, containing data for dropped columns: 0, zstd compressed: 320
```

Алгоритм сжатия по умолчанию - **zstd**.

9) Оценим эффективность выборки из таблиц. Соберите статистику для оптимизатора на таблицы и включите вывод времени выполнения команд:

```
postgres=# VACUUM ANALYZE perf_columnar;
VACUUM
postgres=# VACUUM ANALYZE perf_row;
VACUUM
postgres=# \timing on
Секундомер включён.
```

10) Выполните команды выборки данных из таблиц:

```
postgres=# SELECT vendor_id, SUM(quantity) FROM perf_row GROUP BY vendor_id OFFSET 1000;
 vendor_id | sum
```

```
-----+-----
```

(0 строк)

Время: 105.842 мс

```
postgres=# SELECT vendor_id, SUM(quantity) FROM perf_columnar GROUP BY vendor_id OFFSET 1000;
```

```
 vendor_id | sum
```

```
-----+-----
```

(0 строк)

Время: 113.612 мс

Команды используют полное сканирование и индекс не нужен. При выборке из `perf_row` может использоваться распараллеливание. При работе с `perf_columnar` используется план без распараллеливания.

1) Сравните скорость выполнения запросов:

```
postgres=# explain (analyze, verbose, buffers) select ts from perf_row
where ts < '2024-01-01 10:00:00'::timestamp with time zone and ts > '2024-01-01
10:00:05'::timestamp with time zone;
```

QUERY PLAN

```
-----
```

Gather (cost=1000.00..25719.10 rows=1 width=8) (actual time=97.565..100.336 rows=0 loops=1)

Output: ts

Workers Planned: 2

Workers Launched: 2

Buffers: shared hit=12583 read=9636

-> Parallel Seq Scan on public.perf_row (cost=0.00..24719.00 rows=1 width=8) (actual time=38.320..38.32

Output: ts

Filter: ((perf_row.ts < '2024-01-01 10:00:00+03'::timestamp with time zone) AND (perf_row.ts > '202

Rows Removed by Filter: 133333

Buffers: shared hit=12583 read=9636

Worker 0: actual time=0.004..0.007 rows=0 loops=1

Worker 1: actual time=31.721..31.724 rows=0 loops=1

Buffers: shared hit=5808 read=3796

Planning:

Buffers: shared hit=5 dirtied=2

Planning Time: 0.046 ms

Execution Time: 65.014 ms

(16 rows)

Time: 65.390 ms

```
postgres=# explain (analyze, verbose, buffers) select ts from perf_columnar where ts <
'2024-01-01 10:00:00'::timestamp with time zone and ts > '2024-01-01 10:00:05'::timestamp
with time zone;
```

QUERY PLAN

```
-----
```

Gather (cost=1000.00..1057.72 rows=1 width=8) (actual time=30.429..39.952 rows=0 loops=1)

Output: ts

Workers Planned: 2

Workers Launched: 2

Buffers: shared hit=449

-> Parallel Custom Scan (ColumnarScan) on public.perf_columnar (cost=0.00..57.62 rows=166667 width=8) (ac

tual time=0.598..0.602 rows=0 loops=3)

Output: ts

Columnar Projected Columns: ts

Columnar Chunk Group Filters: ((ts < '2024-01-01 10:00:00+03'::timestamp with time zone) AND (ts > '2

024-01-01 10:00:05+03'::timestamp with time zone))

Columnar Chunk Groups Removed by Filter: 39

Columnar Vectorized Filter: ((ts < '2024-01-01 10:00:00+03'::timestamp with time zone) AND (ts > '202

4-01-01 10:00:05+03'::timestamp with time zone))

Buffers: shared hit=449

Worker 0: actual time=0.393..0.397 rows=0 loops=1

Buffers: shared hit=142

Worker 1: actual time=0.339..0.343 rows=0 loops=1

Buffers: shared hit=142

Planning Time: 0.202 ms

Execution Time: 40.108 ms

(21 rows)

Time: 40.667 ms

Ускорение несущественное - в 1,5 раза.

12) Отключите распараллеливание и выполните запрос:

```
postgres=# set parallel_setup_cost = 10000000000000000;
SET
postgres=# explain (analyze, verbose, buffers) select ts from perf_columnar where
ts < '2024-01-01 10:00:00'::timestamp with time zone and ts > '2024-01-01
10:00:05'::timestamp with time zone;
-----
QUERY PLAN
-----
Custom Scan (ColumnarScan) on public.perf_columnar (cost=0.00..40138.28 rows=400000 width=8) (actual
time=0.894..0.898 rows=0 loops=1)
  Output: ts
  Columnar Projected Columns: ts
  Columnar Chunk Group Filters: ((ts < '2024-01-01 10:00:00+03'::timestamp with time zone) AND (ts > '2024-01-
01 10:00:05+03'::timestamp with time zone))
  Columnar Chunk Groups Removed by Filter: 39
  Columnar Vectorized Filter: ((ts < '2024-01-01 10:00:00+03'::timestamp with time zone) AND (ts > '2024-01-01
10:00:05+03'::timestamp with time zone))
  Buffers: shared hit=165
Query Identifier: -4015271170621366384
Planning:
  Buffers: shared hit=85
  Planning Time: 0.198 ms
  Execution Time: 0.999 ms
(12 rows)
```

Time: 1.489 ms

Ускорение значительное - в 40 раз.

13) Проверьте, может быть у люичной таблицы без распараллеливания запрос выполнится тоже быстрее:

```
postgres=# explain (analyze, verbose, buffers) select ts from perf_row
where ts < '2024-01-01 10:00:00'::timestamp with time zone and ts > '2024-01-01
10:00:05'::timestamp with time zone;
-----
QUERY PLAN
-----
Seq Scan on public.perf_row (cost=0.00..28218.00 rows=1 width=8) (actual time=93.047..93.051 rows=0 loops=1)
  Output: ts
  Filter: ((perf_row.ts < '2024-01-01 10:00:00+03'::timestamp with time zone) AND (perf_row.ts > '2024-01-01
10:00:05+03'::timestamp with time zone))
  Rows Removed by Filter: 400000
  Buffers: shared hit=14027 read=8191
  Planning Time: 0.051 ms
  Execution Time: 93.232 ms
(7 rows)
```

Time: 93.613 ms

Запрос к обычной таблице без распараллеливания выполняется в 1,5 раза медленнее (93.232 ms вместо 65.014 ms) и планировщик не ошибся, использовав параллельный план. Планировщик ошибся при работе с колоночной таблицей на небольшом объеме тестовых данных.

12) Удалите таблицы:

```
drop table if exists perf_row;
drop table if exists perf_columnar;
```

Часть 7b. Сравнение алгоритмов сжатия

1) Пересоздайте таблицы:

```
drop table if exists perf_row;
create table perf_row
( id int, name varchar(15), number int, time timestamp, text1 varchar(64)
) WITH (fillfactor = 100);
drop table if exists perf_columnar;
create table perf_columnar
( id int, name varchar(15), number int, time timestamp, text1 varchar(64)
) USING COLUMNAR;
```

2) Заполните таблицу perf_row данными:

```
DO $$
DECLARE
    names varchar(10)[7] := '{"Олег", "Виктория", "Александр", "Семён", "Эмиль",
"Вадим", "Ангелика"}';
    n int;
    interv varchar(20);
BEGIN
    for i in 0..5e5 loop n:=trunc(random()*1000+1);
        interv := n||' days';
        insert into perf_row values( i, names[floor((random()*7))+1::int]
, n
, current_timestamp + interv::interval
, md5(i::text)
);
    end loop;
END$$;
```

Time: 42184.515 ms (00:42.185)

3) Соберите статистику:

```
ANALYZE perf_row;
```

4) Выполните референсный запрос на обычной таблице:

```
select id,name,number from perf_row where id = 50;
select sum(number), avg(id) from perf_row where id between 777 and 7777777;
```

```
 id | name | number
----+-----+-----
 50 | Семён |    477
(1 row)
```

Time: 59.418 ms

```
 sum | avg
-----+-----
250001901 | 250388.500000000000
(1 row)
```

Time: 78.259 ms

5) Создайте индекс и выполните запрос с использованием индекса:

```
create index i on perf_row(id);
select id,name,number from perf_row where id = 50;
```

```
CREATE INDEX
```

Time: 310.441 ms

```
 id | name | number
```

```
-----+-----+-----
50 | Семён | 477
(1 row)
```

Time: 0.506 ms

6) Заполните данными таблицу `perf_columnar`:

```
INSERT INTO perf_columnar SELECT * FROM perf_row;
ANALYZE perf_columnar;
INSERT 0 500001
Time: 1080.677 ms (00:01.081)
ANALYZE
Time: 317.866 ms
```

7) Выполните запросы, позволяющие оценить эффективность хранения и скорость

выполнения двух запросов:

```
SELECT pg_total_relation_size('perf_row')::numeric /
pg_total_relation_size('perf_columnar');
?column?
```

```
-----
4.5606444053895723
(1 строка)
```

```
select sum(number), avg(id) from perf_columnar where id between 777 and 7777777;
```

```
sum | avg
-----+-----
250001901 | 250388.500000000000
(1 row)
```

Time: 40.308 ms

Время выполнения запросов немного быстрее, чем было в запросе к `heap table` в 4 пункте:

78.259 ms.

8) Изменяя алгоритм сжатия, можно повторить команды:

```
TRUNCATE perf_columnar;
select columnar.alter_columnar_table_set('perf_columnar', compression => 'pglz');
INSERT INTO perf_columnar SELECT * FROM perf_row;
SELECT pg_total_relation_size('perf_row')::numeric /
pg_total_relation_size('perf_columnar');
select sum(number), avg(id) from perf_columnar where id between 777 and 7777777;
```

```
TRUNCATE perf_columnar;
select columnar.alter_columnar_table_set('perf_columnar', compression => 'lz4');
INSERT INTO perf_columnar SELECT * FROM perf_row;
SELECT pg_total_relation_size('perf_row')::numeric /
pg_total_relation_size('perf_columnar');
select sum(number), avg(id) from perf_columnar where id between 777 and 7777777;
```

```
INSERT 0 500001
Time: 1498.693 ms (00:01.499)
?column?
```

```
-----
2.7357293868921776
(1 row)
```

```
sum | avg
```

```
-----+-----
250157031 | 250388.500000000000
(1 row)
```

Time: 56.316 ms

```
INSERT 0 500001
Time: 911.776 ms
      ?column?
```

```
-----
2.5776892430278884
(1 row)
```

```
      sum      |      avg
-----+-----
250157031 | 250388.500000000000
(1 row)
```

Time: 45.744 ms

С алгоритмом сжатия `zstd`: размер в 4.56 раза меньше; время выполнения запроса 40.308 ms.

С алгоритмом сжатия `pglz`: 2.73; и время 56.316 ms.

С алгоритмом сжатия `lz4`: 2.57; и время 45.744 ms.

Время вставки у `lz4` - 911.776 ms, у `zstd` - 1080.677 ms, у `pglz` - 1498.693 ms.

Алгоритм сжатия `zstd`, который используется по умолчанию, является наиболее эффективным по уровню сжатия и скорости выборки из таблицы. По скорости вставки `lz4` и `zstd` сравнимы.

Часть 7с. Функционал расширения

1) Посмотрим, что удалять и менять строки можно. Выполните команды:

```
postgres=# delete from perf_columnar where id=0;
DELETE 1
Time: 8.221 ms
postgres=# update perf_columnar set id=0 where id=0;
UPDATE 1
Time: 9.443 ms
```

Ошибкок нет. Расширение pg_columnar на основе Hydra позволяет удалять и обновлять строки.

2) Псевдостолбцы **CTID**, xmin, xmax присутствуют в таблицах с форматом хранения heap и отсутствуют в таблицах с форматом columnar.

xmin - номер транзакции (xid), создавшей строку.

ctid - значение типа tid (Tuple ID, идентификатор строки), которое представляет физический адрес строки, состоит из номера блока данных и номера слота (записи в списке указателей в заголовке блока) внутри блока.

Посмотрите описание типа данных **tid**:

```
postgres=# \dT tid
                Список типов данных
  Схема      | Имя | Описание
-----+-----+-----
 pg_catalog | tid | (block, offset), physical location of tuple
(1 строка)
```

```
postgres=# \x
Расширенный вывод включён.
postgres=# \dT+ tid
Список типов данных
-[ RECORD 1 ]-----+-----
Схема          | pg_catalog
Имя            | tid
Внутреннее имя | tid
Размер       | 6
Элементы       |
Владелец       | postgres
Права доступа  |
Описание       | (block, offset), physical location of tuple
```

```
postgres=# \x
Расширенный вывод выключен.
```

Размерность tid - шесть байт. Четыре байта на номер страницы, два байта на номер слота в заголовке блока. Четыре байта могут адресовать $2^{32}-2=0xFFFFFFFFE$ блоков, что соответствует 32 ТБ (и минус 2 байта) для 8 КБ блока, являющееся ограничением на размер таблицы.

Это ограничение определено в файле исходного кода src/include/storage/block.h как `#define MaxBlockNumber ((BlockNumber) 0xFFFFFFFFE)`.

Таблица (и другие объекты) хранится в файлах размером до 2 ГБ, номер блока указывается относительно первого блока первого файла, нумерация блоков начинается с **нуля**: `ctid=(0,*)`.

Командой `\dt+` можно узнать размер физического места, которое занимают поля типов данных небольшого размера. Например `date`, `boolean`, `timestamp`, `timestampz`, `point`.

3) В `heap` таблице присутствуют псевдостолбцы:

```
postgres=# select ctid, xmin, xmax, * from perf_row where id=3;
 ctid | xmin | xmax | id | name | number | time | text1
-----+-----+-----+----+-----+-----+-----+-----
(0,1) | 1006 | 0 | 0 | Анжелика | 962 | 2026-12-08 15:39:59.029462 | cfcd20849..
(1 строка)
```

4) В `columnar` таблице псевдостолбцы `xmin`, `xmax` отсутствуют, но `ctid` присутствует:

```
postgres=# select ctid, * from perf_columnar where id=3;
 ctid | xmin | xmax | id | name | number | time | text1
-----+-----+-----+----+-----+-----+-----+-----
(0,1) | 1006 | 0 | 0 | Анжелика | 962 | 2026-12-08 15:39:59.029462 | cfcd20849..
(1 строка)
```

Time: 54.794 ms

```
postgres=# select xmin, xmax, * from perf_columnar where id=3;
ERROR: MIN /MAX TransactionID or CommandID not supported for ColumnarScan
Time: 0.648 ms
```

Приложения псевдостолбцами не пользуются. Псевдостолбец `ctid` может использоваться при диагностике ошибок.

5) Посмотрим, что можно использовать ограничения целостности. **Ограничения целостности** `PRIMARY KEY` и `UNIQUE` используют **индекс** для быстрой проверки соответствия вставляемой строки ограничению. По умолчанию автоматически создаётся **уникальный индекс**. `PRIMARY KEY` отличается от `UNIQUE` тем, что добавляет ограничение целостности `NOT NULL` на столбцы, которые указаны в `PRIMARY KEY` ("ключевые столбцы"). При удалении ограничения целостности индекс, используемый ограничением целостности, удаляется. Создание индекса может быть ресурсоемким и долгим, администраторам баз данных нужно знать эти особенности при удалении или добавлении ограничений целостности.

```
postgres=# alter table perf_columnar alter column id drop not null;
ALTER TABLE
postgres=# alter table perf_columnar add unique (id) deferrable;
ERROR: Foreign keys and AFTER ROW triggers are not supported for columnar tables
ПОДСКАЗКА: Consider an AFTER STATEMENT trigger instead.
```

Ограничения целостности с отложенной проверкой (при фиксации транзакции) не поддерживаются.

```
postgres=# alter table perf_columnar add unique (id);
ALTER TABLE
postgres=# \d perf_columnar
Таблица "public.perf_columnar"
Столбец | Тип | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----
id | integer | | | |
name | character varying(15) | | | |
number | integer | | | |
time | timestamp without time zone | | | |
text1 | character varying(64) | | | |
Индексы:
"perf_columnar_id_key" UNIQUE CONSTRAINT, btree (id)
```

Названия индекса и ограничения целостности можно указать в команде, но автоматически создаваемое имя интуитивно понятно.

```
postgres=# \d perf_columnar_id_key
        Индекс "public.perf_columnar_id_key"
        Столбец | Тип      | Ключевой? | Определение
-----+-----+-----+-----
        id      | integer  | да         | id
        УНИКАЛЬНЫЙ, btree, для таблицы "public.perf_columnar"
postgres=# alter table perf_columnar drop constraint perf_columnar_id_key;
ALTER TABLE
postgres=# alter table perf_columnar add primary key (id);
ALTER TABLE
postgres=# \d perf_columnar
        Таблица "public.perf_columnar"
        Столбец | Тип                                     | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----
        id      | integer                                |                     | not null           |
        name    | character varying(15)                 |                     |                    |
        number  | integer                                |                     |                    |
        time    | timestamp without time zone           |                     |                    |
        text1   | character varying(64)                 |                     |                    |
Индексы:
        "perf_columnar_pkey" PRIMARY KEY, btree (id)
```

6) Проверим, используется ли индекс:

```
postgres=# explain select id from perf_columnar where id = 10000;
        QUERY PLAN
-----
        Index Scan using perf_columnar_pkey on perf_columnar  (cost=0.42..347.05 rows=1 width=4)
        Index Cond: (id = 10000)
(2 rows)
```

Индекс **используется**.

7) Удалите первичный ключ:

```
postgres=# alter table perf_columnar drop constraint perf_columnar_pkey;
ALTER TABLE
postgres=# \d perf_columnar
        Таблица "public.perf_columnar"
        Столбец | Тип                                     | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----
        id      | integer                                |                     | not null           |
        name    | character varying(15)                 |                     |                    |
        number  | integer                                |                     |                    |
        time    | timestamp without time zone           |                     |                    |
        text1   | character varying(64)                 |                     |                    |
```

При удалении ограничения целостности удаляется индекс, используемый ограничением целостности. Ограничение целостности **NOT NULL** не удаляется, так как в системном каталоге не сохраняется, существовало ли оно до создания ограничения целостности, или было добавлено при создании ограничения целостности типа `PRIMARY KEY`.

8) В таблицу можно вставлять строки. Кроме команды `INSERT` можно использовать команду `COPY`. Выполните команду:

```
postgres=# COPY perf_columnar (id) FROM PROGRAM 'echo 500001';
COPY 1
```

Команда успешно вставила одну строку.

9) Посмотрите конфигурационные параметры расширения:

```
postgres=# \dconfig columnar.*
      Parameter | Value
-----+-----
columnar.chunk_group_row_limit | 10000
columnar.column_cache_size     | 200MB
columnar.compression           | zstd
columnar.compression_level     | 3
columnar.enable_column_cache   | off
columnar.min_parallel_processes | 8
columnar.planner_debug_level   | debug3
columnar.stripe_row_limit      | 150000
(8 rows)
```

Если список пуст, то это означает что в текущей сессии не использовался функционал расширения. В этом терминале можно дать команду, которая задействует функционал расширения. Например:

```
select id,name from perf_columnar where id = 3;
```

10) Посмотрите, какие значения можно установить для алгоритма сжатия:

```
postgres=# set columnar.compression TO <TAB><TAB>
DEFAULT lz4      "none"  pglz   zstd
```

Поддерживается три алгоритма сжатия.

11) Расширение создаёт представление:

```
postgres=# select * from columnar.options;
 regclass | chunk_group_row_limit | stripe_row_limit | compression_level | compression
-----+-----+-----+-----+-----
perf_columnar | 10000 | 150000 | 3 | zstd
(1 row)
```

Time: 0.481 ms

Параметры на уровне таблиц можно устанавливать функцией:

```
postgres=# select columnar.alter_columnar_table_set('perf_columnar', compression
=> 'lz4');
```

```
alter_columnar_table_set
-----
```

(1 row)

```
postgres=# select * from columnar.options;
 regclass | chunk_group_row_limit | stripe_row_limit | compression_level | compression
-----+-----+-----+-----+-----
perf_columnar | 10000 | 150000 | 3 | lz4
(1 row)
```

12) Удалите таблицы:

```
postgres=# drop table perf_row;
drop table perf_columnar;
DROP TABLE
DROP TABLE
```

Глава 5. Журналирование

Часть 1. Какая информация попадает в журнал

Запустите psql:

```
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
```

```
postgres=#
```

```
postgres=# SHOW log_line_prefix;
log_line_prefix
```

```
-----
%m [%p:%v] [%d] %r %a
(1 row)
```

%m: Уровень сообщения (DEBUG5, DEBUG4, INFO, WARNING, ERROR, и так далее).

[%p:%v]: Идентификатор процесса PostgreSQL и номер версии протокола.

[%d]: Имя базы данных.

%r: Идентификатор транзакции.

%a: IP-адрес и порт клиента.

Часть 2. Расположение журналов сервера

1) Посмотрим путь до журналов:

```
postgres=# SHOW log_directory;
log_directory
-----
log
(1 row)
```

По умолчанию установлено как поддиректория относительно PGDATA.
Какая маска у файлов журнала?

```
postgres=# SHOW log_filename;
log_filename
-----
postgresql-%Y-%m-%d_%H%M%S.log
(1 row)
```

Где находится PGDATA?

```
postgres=# SHOW data_directory;
data_directory
-----
/var/lib/postgresql/tantor-se-17/data
(1 row)
```

Включен ли logger?

```
postgres=# show logging_collector;
logging_collector
-----
off
(1 строка)
```

Не включён и журнал направляется в операционную систему.

Включим logging collector и поменяем формат имени файла диагностического журнала:

```
postgres=# alter system set logging_collector = on;
alter system set log_filename = 'postgresql-%F.log';
ALTER SYSTEM
postgres=# \q
postgres@tantor:~$ sudo systemctl restart tantor-se-server-17
postgres@tantor:~$ psql
```

2) Посмотрим содержимое папки журнала:

```
postgres=# \! ls -l $PGDATA/log

total 148228
-rw----- 1 postgres postgres      1115 Jun  25  2025 postgresql-2025-07-25.log
```

3) Посмотрим содержимое файла журнала:

```
postgres=# \! tail -n 10 $PGDATA/log/postgresql-*
[33452] LOG:  starting Tantor Special Edition 17.5.0 8205c5ba on x86_64-pc-linux-g
nu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
[33452] LOG:  listening on IPv4 address "127.0.0.1", port 5432
[33452] LOG:  listening on IPv6 address ":::1", port 5432
[33452] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
[33456] LOG:  database system was shut down at ...
[33452] LOG:  database system is ready to accept connections
```

Часть 3. Как информация попадает в журнал

```
postgres=# CREATE TABLE t (id integer);
CREATE TABLE
postgres=# \! tail -n 10 $PGDATA/log/postgresql-*
```

```
[5289:8/30] [postgres] [local] psql LOG:  statement: create table t (id integer);
```

Часть 4. Добавление формата csv

1) Посмотрим параметр:

```
postgres=# SHOW log_destination;
log_destination
-----
stderr
(1 row)
```

2) Изменим параметр и перечитаем конфигурацию:

```
postgres=# ALTER SYSTEM SET log_destination = stderr, csvlog ;
ALTER SYSTEM
```

```
postgres=# SELECT pg_reload_conf();
pg_reload_conf
-----
t
(1 row)
```

3) Посмотрим, что параметр успешно применяется.

```
postgres=# SHOW log_destination;
log_destination
```

```
-----
stderr, csvlog
(1 row)
```

4) Вставим новое значение в таблицу t:

```
postgres=# INSERT INTO t VALUES (1);
INSERT 0 1
```

5) Посмотрим содержимое файла:

```
postgres=# \! ls -l $PGDATA/log
-rw----- 1 postgres postgres      1115 Jun 25 2025 postgresql-2025-07-25.log
-rw----- 1 postgres postgres      1115 Jun 25 2025 postgresql-2025-07-25.csv
```

6) Добавился формат данных csv:

```
postgres=# \! tail -n 10 $PGDATA/log/postgres*.csv
08:08:54.580
MSK,"postgres","postgres",9199,"[local]",65e01024.23ef,3,"idle",08:03:32
MSK,5/325,0,LOG,00000,"statement: insert into t
values(1);",,,,,,,,,,"psql","client backend",,0
```

7) Сравним с содержимым обычного журнала:

```
postgres=# INSERT INTO t VALUES (1);
INSERT 0 1

postgres=# \! tail -n 1 /var/lib/postgresql/tantor-se-17/data/log/p*.log
[9199:5/326] [postgres] [local] psql LOG: statement: insert into t values(1);
postgres=#
```

Удалим ненужные объекты:

```
postgres=# DROP TABLE t;
DROP TABLE

postgres=# ALTER SYSTEM SET log_destination = stderr ;
ALTER SYSTEM

postgres=# SELECT pg_reload_conf();
 pg_reload_conf
-----
t
(1 row)
```

Глава 6. Безопасность

Часть 1. Создание новой роли

1) Запустите инструмент `psql`:

```
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
```

```
postgres=#
```

2) Создадим новую роль:

```
postgres=# CREATE ROLE user1;
CREATE ROLE
```

3) Посмотрим, какие есть роли в кластере:

```
postgres=# \du
                                List of roles
Role name |                               Attributes
-----+-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
user1     | Cannot login
```

Часть 2. Установка атрибутов

```
postgres=# ALTER ROLE user1 LOGIN CREATEDB;
ALTER ROLE
```

```
postgres=# \du
                                List of roles
Role name |                               Attributes
-----+-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
user1     | Create DB
```

Часть 3. Создание групповой роли

Предположим, что нам нужна роль, под которой можно только подключаться к кластеру, а под второй - создавать БД, но нельзя делать соединения к БД.

1) Создадим вторую роль:

```
postgres=# CREATE USER user2;
CREATE ROLE
```

2) Уберём право создания соединений:

```
postgres=# ALTER ROLE user1 NOLOGIN;
ALTER ROLE
```

```
postgres=# \du
                                List of roles
Role name |                               Attributes
-----+-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
user1     | Create DB, Cannot login
```

```
user2 |
```

3) Включим пользователя user2 в роль user1.

```
postgres=# GRANT user1 TO user2;
GRANT ROLE
```

4) Проверим результат:

```
postgres=# \drg
                List of role grants
Role name | Member of | Options | Grantor
-----+-----+-----+-----
user2    | user1    | INHERIT, SET | postgres
(1 row)
```

5) Первая роль не может подсоединиться:

```
postgres=# \c - user1
connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL:  role
"user1" is not permitted to log in
Previous connection kept
```

6) Входим под второй ролью:

```
postgres=# \c - user2
You are now connected to database "postgres" as user "user2".
```

7) Пытаемся создать базу данных под второй ролью:

```
postgres=> CREATE DATABASE dat1;
ERROR:  permission denied to create database
```

8) Переключаем роль на первую:

```
postgres=> SET ROLE user1;
SET
```

9) Теперь создать базу данных можно:

```
postgres=> CREATE DATABASE dat1;

CREATE DATABASE
```

10) Вернемся к роли user2:

```
postgres=> RESET ROLE;
RESET
```

11) Подключаемся к БД dat1:

```
dat1=> \c dat1
You are now connected to database "dat1" as user "user2".
```

Часть 4. Создание схемы и таблицы

```
dat1=> CREATE SCHEMA sch1;
CREATE SCHEMA
```

Посмотрим, кто владелец схемы:

```
dat1=> \dn+
                List of schemas
Name | Owner | Access privileges | Description
-----+-----+-----+-----
public | pg_database_owner | pg_database_owner=UC/pg_database_owner+ | standard public schema
```

```

sch1 | user2 | =U/pg_database_owner |
(2 строки)
    
```

```

dat1=> CREATE TABLE sch1.a1 (id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
str text);
CREATE TABLE
    
```

Посмотрим описание таблицы:

```
dat1=> \d sch1.a1
```

```

          Таблица "sch1.a1"
  Столбец | Тип | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----
id       | integer | | not null | generated always as identity
str      | text | | |
    
```

Индексы:

```
"a1_pkey" PRIMARY KEY, btree (id)
```

Посмотрим разрешения на таблицу:

```
dat1=> \dp sch1.a1
```

```

          Права доступа
  Схема | Имя | Тип | Права доступа | Права для столбцов | Политики
-----+-----+-----+-----+-----+-----
sch1   | a1  | таблица | | |
(1 строка)
    
```

Пока нет ни у какой роли, кроме суперпользовательской.

Часть 5. Выдача роли доступа к таблице

1) Создадим еще одну роль:

```
dat1=> \c - postgres
```

Вы подключены к базе данных "dat1" как пользователь "postgres"

```

dat1=# CREATE ROLE user3 LOGIN;
CREATE ROLE
    
```

2) Попробуем получить доступ к таблице a1:

```
dat1=# \c - user3
```

Вы подключены к базе данных "dat1" как пользователь "user3".

```
dat1=> \dn
```

```

          Список схем
  Имя | Владелец
-----+-----
public | pg_database_owner
sch1   | user2
    
```

(2 rows)

```
dat1=> SELECT * FROM sch1.a1;
```

```

ОШИБКА: нет доступа к схеме sch1
СТРОКА 1: SELECT * FROM sch1.a1;
    
```

3) В доступе отказано - нет привилегий на схему:

```
dat1=> \c - postgres
```

Вы подключены к базе данных "dat1" как пользователь "postgres"

```
dat1=> GRANT USAGE on SCHEMA sch1 TO user3;
GRANT
```

```
dat1=> \dn+ sch1
```

```

                Список схем
Имя  | Владелец | Права доступа | Описание
-----+-----+-----+-----
sch1 | user2   | user2=UC/user2+|
      |         | user3=U/user2  |
(1 строка)

```

```
dat1=> \c - user3
```

```
You are now connected to database "dat1" as user "user3".
```

```
dat1=> SELECT * FROM sch1.a1;
```

```
ОШИБКА: нет доступа к таблице a1
```

Теперь отказ из-за отсутствия привилегий на таблице a1:

```
dat1=> \c - postgres
```

```
Вы подключены к базе данных "dat1" как пользователь "postgres"
```

```
dat1=> GRANT SELECT, INSERT (str) ON TABLE sch1.a1 to user3;
```

```
GRANT
```

```
dat1=> \dp sch1.a1
```

```

                Права доступа
Схема | Имя | Тип | Права доступа | Права для столбцов | Политики
-----+-----+-----+-----+-----+-----
sch1  | a1  | таблица | user2=arwdDxt/user2+| str:                +|
      |    |         | user3=r/user2      | user3=a/user2      |
(1 строка)

```

```
dat1=> \c - user3
```

```
Вы подключены к базе данных "dat1" как пользователь "user3"
```

```
dat1=> SELECT * FROM sch1.a1;
```

```
id | str
----+-----
(0 строк)
```

Теперь все в порядке. Доступ предоставлен в рамках выданных привилегий.

Проверим вставку в столбец:

```
dat1=> INSERT INTO sch1.a1 (str) VALUES ('первая запись');
INSERT 0 1
```

```
dat1=> SELECT * FROM sch1.a1;
```

```
id | str
----+-----
 1 | первая запись
(1 row)
```

Проверим вставку в первый столбец:

```
dat1=> INSERT INTO sch1.a1 OVERRIDING SYSTEM VALUE values (2);
```

```
ОШИБКА: нет доступа к таблице a1
```

Не хватает привилегий.

Удаление также строк и объекта невозможно - нужно быть владельцем или суперпользователем:

```
dat1=> DELETE FROM sch1.a1;
ОШИБКА: нет доступа к таблице a1
dat1=> DROP TABLE sch1.a1;
ОШИБКА: нужно быть владельцем таблицы a1
```

Часть 6. Удаление созданных объектов

Удалим схему:

```
dat1=> \c - user2
Вы подключены к базе данных "dat1" как пользователь "user2".
```

```
dat1=> DROP SCHEMA sch1;
ОШИБКА: удалить объект схема sch1 нельзя, так как от него зависят другие объекты
ПОДРОБНОСТИ: таблица sch1.a1 зависит от объекта схема sch1
ПОДСКАЗКА: Для удаления зависимых объектов используйте DROP ... CASCADE.
```

Схема не пуста, можно выполнить каскадное удаление:

```
dat1=> DROP SCHEMA sch1 CASCADE;
ЗАМЕЧАНИЕ: удаление распространяется на объект таблица sch1.a1
DROP SCHEMA
```

Переключимся на другую базу данных и удалим **dat1**:

```
dat1=> \c postgres
Вы подключены к базе данных "postgres" как пользователь "user2".
```

```
postgres=> DROP DATABASE dat1 (force);
DROP DATABASE
```

Для удаления ролей воспользуемся суперпользовательской ролью:

```
postgres=> \c - postgres
Вы подключены к базе данных "postgres" как пользователь "postgres".
```

```
postgres=# DROP ROLE user1, user2, user3;
DROP ROLE
```

Подключение и аутентификация

Часть 1. Расположение файлов конфигурации

1) Запустите psql:

```
astra@tantor:~$ sudo su - postgres
```

```
postgres@tantor:~$ psql
```

```
psql (17.5)
```

Введите "help", чтобы получить справку.

2) Посмотрим месторасположение конфигурационного файла:

```
postgres=# SHOW hba_file;
```

```
hba_file
```

```
-----  
/var/lib/postgresql/tantor-se-17/data/pg_hba.conf
```

```
(1 строка)
```

3) Можно посмотреть правила подключения с помощью представления

pg_hba_file_rules:

```
postgres=# \d pg_hba_file_rules;
```

```
Представление "pg_catalog.pg_hba_file_rules"
```

Столбец	Тип	Правило сортировки	Допустимость NULL	По умолчанию
rule_number	integer			
file_name	text			
line_number	integer			
type	text			
database	text[]			
user_name	text[]			
address	text			
netmask	text			
auth_method	text			
options	text[]			
error	text			

Часть 2. Локальные изменения для аутентификации

1) Текстовым редактором:

```
postgres@tantor:~$ mcedit /var/lib/postgresql/tantor-se-17/data/pg_hba.conf
```

добавьте строку:

```
postgres@tantor:~$ tail -n 14 /var/lib/postgresql/tantor-se-17/data/pg_hba.conf
```

```
# TYPE DATABASE USER ADDRESS METHOD
local postgres astra peer map=map1
# "local" is for Unix domain socket connections only
local all all trust
# IPv4 local connections:
host all all 127.0.0.1/32 trust
# IPv6 local connections:
host all all ::1/128 trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
local replication all trust
host replication all 127.0.0.1/32 trust
host replication all ::1/128 trust
```

2) Текстовым редактором:

```
postgres@tantor:~$ mcedit /var/lib/postgresql/tantor-se-17/data/pg_ident.conf
```

добавьте строку:

```
postgres@tantor:~$ tail -n 3 /var/lib/postgresql/tantor-se-17/data/pg_ident.conf
```

```
# MAPNAME          SYSTEM-USERNAME    PG-USERNAME
map1               astra              user1
```

3) Перечитайте конфигурацию:

```
postgres@tantor:~$ pg_ctl reload
server signaled
```

4) Создайте двух пользователей user1 и user2:

```
postgres@tantor:~$ psql
```

```
psql (17.5)
Введите "help", чтобы получить справку
```

```
postgres=# CREATE USER user1;
CREATE ROLE
postgres=# CREATE ROLE user2 LOGIN;
CREATE ROLE
```

```
postgres=# \du
                                List of roles
Role name |                               Attributes
-----+-----
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
user1    |
user2    |
```

Оба пользователя имеют атрибут LOGIN.

3) Посмотрим, есть ли ошибки в конфигурации:

```
postgres=# \d pg_ident_file_mappings;
                                Представление "pg_catalog.pg_ident_file_mappings"
  Столбец   |  Тип   | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----
map_number | integer |                    |                    |
file_name  | text    |                    |                    |
line_number | integer |                    |                    |
map_name   | text    |                    |                    |
sys_name   | text    |                    |                    |
pg_username | text    |                    |                    |
error      | text    |                    |                    |
```

```
postgres=# SELECT map_number, line_number, map_name, sys_name, pg_username, error
FROM pg_ident_file_mappings;
map_number | line_number | map_name | sys_name | pg_username | error
-----+-----+-----+-----+-----+-----
          1 |           73 | map1    | astra   | user1      |
(1 row)
```

В столбце error пусто, значит ошибок нет.

```
postgres=# SELECT rule_number, type, database, user_name, auth_method, address, options, error
FROM pg_hba_file_rules();
rule_number | type | database | user_name | auth_method | address | options | error
-----+-----+-----+-----+-----+-----+-----+-----
          1 | local | {postgres} | {astra} | peer       |        | {map=map1} |
          2 | local | {all}     | {all}    | trust     |        |          |
```

```

3 | host | {all} | {all} | trust | 127.0.0.1 | |
4 | host | {all} | {all} | trust | ::1 | |
5 | local | {replication} | {all} | trust | | |
6 | host | {replication} | {all} | trust | 127.0.0.1 | |
7 | host | {replication} | {all} | trust | ::1 | |
(7 rows)
    
```

В столбце error пусто, значит нет ошибок нет.

5) В терминале пользователя **astra** подсоединитесь к базе данных postgres:

```

astra@tantor:~$ psql -U user1 -d postgres
Pager usage is off.
psql (17.5)
Type "help" for help.
    
```

```

postgres=> select current_user;
user
-----
user1
(1 row)
    
```

Сессия создана под пользователем **user1**.

6) Очистка ненужных объектов

```

postgres=> \c postgres postgres
You are now connected to database "postgres" as user "postgres".
postgres=# DROP user user1, user2;
DROP ROLE
    
```

7) Текстовым редактором:

```

postgres@tantor:~$ mcedit /var/lib/postgresql/tantor-se-17/data/pg_hba.conf
    
```

закомментируйте строку:

```

postgres@tantor:~$ tail -n 14 /var/lib/postgresql/tantor-se-17/data/pg_hba.conf

# TYPE DATABASE USER ADDRESS METHOD
#local postgres astra peer map=map1
# "local" is for Unix domain socket connections only
local all all trust
# IPv4 local connections:
host all all 127.0.0.1/32 trust
# IPv6 local connections:
host all all ::1/128 trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
local replication all trust
host replication all 127.0.0.1/32 trust
host replication all ::1/128 trust
    
```

8) Перечитайте конфигурацию:

```

postgres@tantor:~$ pg_ctl reload
server signaled
    
```

```

postgres=# SELECT rule_number, type, database, user_name, auth_method, address,
options, error FROM pg_hba_file_rules();
    
```

rule_number	type	database	user_name	auth_method	address	options
1	local	{all}	{all}	trust		
2	host	{all}	{all}	trust	127.0.0.1	
3	host	{all}	{all}	trust	::1	
4	local	{replication}	{all}	trust		
5	host	{replication}	{all}	trust	127.0.0.1	
6	host	{replication}	{all}	trust	::1	

(6 rows)

Глава 7а. Физическое резервирование

Часть 1. Создание базовой резервной копии кластера

1) Утилита `pg_basebackup` не резервирует, если директория, куда делается бэкап, существует и при этом не пустая. В терминале пользователя `postgres` удалите директорию:

```
postgres@tantor:~$ rm -rf $HOME/backup
```

2) Мы будем резервировать на тот же хост, где располагается резервируемый кластер. Если есть табличные пространства, то нужно будет указывать соответствие (mapping) их директориям. Проверьте? есть ли табличные пространства в кластере:

```
postgres@tantor:~$ ls -l $PGDATA/pg_tblspc
total 0
lrwxrwxrwx 1 postgres postgres 44 Mar 10 13:41 32913 ->
/var/lib/postgresql/tantor-se-17/data/./u01
```

В директории есть символическая ссылка, значит есть табличное пространство.

Табличное пространство создавалось в пунктах 1 и 2 части 2 практики к главе 4b командами:

```
postgres=# \! mkdir /var/lib/postgresql/tantor-se-17/u01
```

```
postgres=#
```

```
CREATE TABLESPACE u01tbs LOCATION '/var/lib/postgresql/tantor-se-17/u01';
```

Если директории и табличного пространства нет, то создайте его этими командами.

3) Создайте таблицу в табличном пространстве `u01tbs`:

```
postgres=# CREATE TABLE t (id bigserial, t text) TABLESPACE u01tbs;
CREATE TABLE
```

4) Создайте бэкап:

```
postgres@tantor:~$
pg_basebackup -D $HOME/backup/1 -T $PGDATA/./u01=$HOME/backup/1/u01 -P -c fast
30302/30302 kB (100%), 2/2 tablespaces
```

5) Посмотрите содержимое резервной копии:

```
postgres@tantor:~$ ls -w 60 $HOME/backup/1
backup_label      pg_multixact      pg_twophase
backup_manifest   pg_notify         PG_VERSION
base              pg_replslot       pg_wal
global            pg_serial          pg_xact
pg_commit_ts      pg_snapshots      postgresql.auto.conf
pg_dynshmem       pg_stat            postgresql.conf
pg_hba.conf       pg_stat_tmp       u01
pg_ident.conf     pg_subtrans
pg_logical         pg_tblspc
```

6) Посмотрите, на какую директорию указывает символическая ссылка табличного пространства:

```
postgres@tantor:~$ ls -l $HOME/backup/1/pg_tblspc
```

```
total 0
lrwxrwxrwx 1 postgres postgres 32 32913 -> /var/lib/postgresql/backup/1/u01
```

Всё корректно: если запустить ещё один экземпляр, используя в качестве PGDATA директорию бэкапа, то директория табличного пространства будет найдена и использована по этому (`/var/lib/postgresql/backup/1/u01`) пути, а не по пути из кластера (`/var/lib/postgresql/tantor-se-17/data/./u01`), который резервировали.

Часть 2. Запуск экземпляра на копии кластера

1) В файле `$HOME/backup/1/postgresql.conf` параметр `port` закомментирован, это значит, что будет использоваться значение по умолчанию 5432. Нужно установить другое значение порта, так как порт 5432 занят экземпляром кластера, который мы резервировали.

Можно использовать любое значение выше 1023 (на портах ниже 1024 процессы непривилегированных пользователей операционной системы не могут прослушивать). Порт не должен быть занят (желательно не занят ни на одном интерфейсе).

Устанавливать порт (как и другие параметры) можно в параметре командной строки, передаваемой процессу `postgres` (в том числе через утилиты-обёртки, например, `pg_ctl`) в `postgresql.auto.conf` или в `postgresql.conf`. Выбирают наиболее удобный способ.

2) Установите в основном файле параметров значение порта 5433:

```
postgres@tantor:~$ echo "port = 5433" >> $HOME/backup/1/postgresql.conf
```

3) Запустите экземпляр:

```
postgres@tantor:~$ pg_ctl start -D $HOME/backup/1
```

ожидание запуска сервера....

СООБЩЕНИЕ: передача вывода в протокол процессу сбора протоколов

ПОДСКАЗКА: В дальнейшем протоколы будут выводиться в каталог "log".

готово

сервер запущен

В лог кластера будут выводиться сообщения:

```
postgres@tantor:~$ tail $HOME/backup/1/log/postgresql-*.log
```

```
LOG: starting PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
LOG: listening on IPv4 address "0.0.0.0", port 5433
LOG: listening on IPv6 address ":::", port 5433
LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5433"
LOG: database system was interrupted; last known up at 23:36:29 MSK
LOG: redo starts at 115/A9000028
LOG: consistent recovery state reached at 115/A9000178
LOG: redo done at 115/A9000178 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG: checkpoint starting: end-of-recovery immediate wait
LOG: checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.003 s,
sync=0.001 s, total=0.008 s; sync files=3, longest=0.001 s, average=0.001 s; distance=16384 kB, estimate=16384
kB; lsn=115/AA000028, redo lsn=115/AA000028
```

4) Прочтите этот пункт, но не выполняйте. Если вы хотите, чтобы диагностические сообщения выводились на экран, то нужно закомментировать в файле `postgresql.conf` строку с параметром `logging_collector = 'on':`

```
postgres@tantor:~$ cat $HOME/backup/1/postgresql.auto.conf
```

```
# Do not edit this file manually!
```

```
# It will be overwritten by the ALTER SYSTEM command.
```

```
shared_preload_libraries = 'pg_stat_statements, pg_qualstats, pg_store_plans,
```

```
pg_prewarm, pg_stat_kcache'
```

```
logging_collector = 'on'
```

```
log_filename = 'postgresql-%F.log'
```

```
log_destination = 'stderr'
```

или добавить строку в файл конфигурации, например, командой:

```
postgres@tantor:~$
```

```
echo "logging_collector = off" >> $HOME/backup/1/postgresql.auto.conf
```

Рестартовать экземпляр и проверить, что сообщения выводятся:

```
postgres@tantor:~$ pg_ctl stop -D $HOME/backup/1
ожидание завершения работы сервера....
готово
сервер остановлен
postgres@tantor:~$ pg_ctl start -D $HOME/backup/1
ожидание запуска сервера....
[20912] СООБЩЕНИЕ:  запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3)
12.2.0, 64-bit
[20912] СООБЩЕНИЕ:  для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5433
[20912] СООБЩЕНИЕ:  для приёма подключений по адресу IPv6 ":::" открыт порт 5433
[20912] СООБЩЕНИЕ:  для приёма подключений открыт Unix-сокет "/var/run/postgresql/.s.PGSQL.5433"
[20915] СООБЩЕНИЕ:  система БД была выключена: MSK
[20912] СООБЩЕНИЕ:  система БД готова принимать подключения
готово
сервер запущен
```

5) Подсоединитесь к экземпляру:

```
postgres@tantor:~$ psql -p 5433
```

Так как экземпляр открылся в обычном режиме ("чтение-запись", режим, позволяющий вносить изменения), мы получили **клон** исходного кластера.

Часть 3. Файлы журнала

1) Посмотрите название текущего файла журнала в сессии к реплике (на порт 5433):

```
postgres=# select pg_walfile_name_offset(pg_current_wal_lsn()),
pg_current_wal_lsn();
 pg_walfile_name_offset | pg_current_wal_lsn
-----+-----
 (0000000100000115000000AA,264) | 115/AA000108
(1 строка)
```

Линия времени не изменилась и равна **1**. Почему?

Потому что мы запустили экземпляр, процесс startup накатил журнальные записи, и для него это выглядело как обычный запуск экземпляра после аварийной остановки.

Почему аварийной?

Потому что мы делали бэкап на работающем кластере, а не на корректно остановленном. Если бы мы остановили кластер, то мы бы не смогли воспользоваться утилитой `pg_basebackup`. Утилита `pg_basebackup` не может делать бэкапы на остановленном кластере.

2) Переключите журнал и посмотрите что изменится, чтобы увидеть как меняются числа:

```
postgres=# select pg_switch_wal();
 pg_switch_wal
-----
 115/AA000122
(1 строка)
```

3) Что выдала функция? LSN в том файле, с которого переключились, плюс один байт. То есть LSN начала неиспользуемой части файла журнала.

Посмотрите какой файл стал текущим:

```
postgres=# select pg_walfile_name_offset(pg_current_wal_lsn()),
pg_current_wal_lsn();
 pg_walfile_name_offset | pg_current_wal_lsn
-----+-----
 (0000000100000115000000AB,112) | 115/AB000070
(1 строка)
```

Значение последнего символа в названии файла увеличилось на единицу. Буквы и цифры в названии файлов журнала представляют собой шестнадцатеричную запись.

4) Выполним функцию переключения файла журнала несколько раз:

```
postgres=# select pg_switch_wal();
 pg_switch_wal
-----
 115/AB00008A
(1 строка)
```

```
postgres=# select pg_switch_wal();
 pg_switch_wal
-----
 115/AC000000
(1 строка)
```

```
postgres=# select pg_switch_wal();
 pg_switch_wal
-----
```

115/AC000000
(1 строка)

5) Почему последние вызовы не переключали журнал? Это описано в документации (https://docs.tantorlabs.ru/tdb/ru/17_5/se/functions-admin.html):

"Если с момента последнего переключения файлов журнала предварительной записи не было никакой активности, `pg_switch_wal` ничего не делает и возвращает начальное положение файла журнала предварительной записи, который в настоящее время используется".

6) Подставляя произвольные значения, убедитесь, что функция `pg_walfile_name_offset` рассчитывает значения в зависимости от линии времени и размера файла журнала:

```
postgres=# select pg_walfile_name_offset('ABCD/EF00FFFF');
           pg_walfile_name_offset
-----
(0000000010000ABCD000000EF, 65535)
(1 строка)
```

Мы посмотрели, как можно по виду LSN (синий цвет значения EF), предположить в каком файле журнала находится этот LSN без вызова функций.

7) Остановите экземпляр клона:

```
postgres@tantor:~$ pg_ctl stop -D $HOME/backup/1
```

Пример сообщений на английском языке:

```
LOG: received fast shutdown request
LOG: aborting any active transactions
LOG: background worker "logical replication launcher" (PID 31666) exited with exit code 1
waiting for server to shut down....
LOG: shutting down
LOG: checkpoint starting: shutdown immediate
LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.001 s, sync=0.001 s, total=0.005 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=0 kB, estimate=44236 kB; lsn=115/AE000198, redo
lsn=115/AE000198
LOG: database system is shut down
done
server stopped
```

Можно сравнить с сообщениями на русском языке.

Далее в этой практике будут приводиться примеры сообщений утилит на английском языке, чтобы вы могли сравнить их с сообщениями на русском языке, которые вы будете получать в консоли.

Часть 4. Проверка целостности резервной копии

1) `pg_basebackup` создала файл `backup_manifest`, с помощью которого можно проверить, не поменялись ли файлы в бэкапе за время их хранения. Проверим копию, на которой уже запускали экземпляр:

```
postgres@tantor:~$ pg_verifybackup $HOME/backup/1
```

```
pg_verifybackup: error: "pg_stat/pg_stat_statements.stat" is present on disk but not in the manifest
pg_verifybackup: error: "pg_stat/pgstat.stat" is present on disk but not in the manifest
pg_verifybackup: error: "postmaster.opts" is present on disk but not in the manifest
pg_verifybackup: error: "base/5/pg_internal.init" is present on disk but not in the manifest
pg_verifybackup: error: "global/pg_internal.init" is present on disk but not in the manifest
pg_verifybackup: error: "global/pg_store_plans.stat" is present on disk but not in the manifest
pg_verifybackup: error: "postgresql.conf" has size 30440 on disk but size 30428 in the manifest
pg_verifybackup: error: "backup_label.old" is present on disk but not in the manifest
pg_verifybackup: error: "pg_subtrans/0000000000001" is present on disk but not in the manifest
pg_verifybackup: error: "u01/Pg_17_642505061/5/382539" is present on disk but not in the manifest
pg_verifybackup: error: "u01/Pg_17_642505061/5/382539.1" is present on disk but not in the manifest
pg_verifybackup: error: "u01/Pg_17_642505061/5/382541" is present on disk but not in the manifest
pg_verifybackup: error: "u01/Pg_17_642505061/5/382540" is present on disk but not in the manifest
pg_verifybackup: error: "u01/Pg_17_642505061/5/382539_vm" is present on disk but not in the manifest
pg_verifybackup: error: "u01/Pg_17_642505061/5/382539_fsm" is present on disk but not in the manifest
pg_verifybackup: error: "backup_label" is present in the manifest but not on disk
pg_waldump: error: could not find file "0000000100000115000000A9": No such file or directory
pg_verifybackup: error: WAL parsing failed for timeline 1
```

Сообщения о файлах `.stat` `internal.init` `pg_subtrans/*` обычны, файлы не попадают в бэкап. `postgresql.conf` мы обновили, добавив номер порта. Файл журнала A9 исчез, потому что не понадобился клону для его восстановления, и не удерживался параметром `min_wal_size`. Файл `backup_label` был переименован в `backup_label.old`

Файл `backup_label` важен, если он присутствует, потому что используются значения из него, для того чтобы определить с какого LSN начинать накат журналов, а не данные из `pg_control`. Содержимое `pg_control` было изменено экземпляром, он присутствует в файле манифеста, если его удалить, то будет выдано сообщение, но `pg_control` не был выдан в списке изменённых.

2) Почему присутствуют записи обо всех файлах табличного пространства?

Контрольные суммы этих файлов есть в `backup_manifest`. Файлы не менялись и были успешно проверены:

```
postgres@tantor:~$ cat $HOME/backup/1/backup_manifest | grep pg_tblspc
```

```
{ "Path": "pg_tblspc/32913/Pg_17_642505061/5/382541", "Size": 8192, "Last-Modified":
"11:16:27 GMT", "Checksum-Algorithm": "CRC32C", "Checksum": "381590e3" },
{ "Path": "pg_tblspc/32913/Pg_17_642505061/5/382540", "Size": 0, "Last-Modified":
"11:16:27 GMT", "Checksum-Algorithm": "CRC32C", "Checksum": "00000000" },
```

Строки об отсутствующих файлах появились потому, что при резервировании директорию табличного пространства поместили в поддиректорию `PGDATA/u01`. Это одна из причин почему **директории табличных пространств стоит располагать вне PGDATA.**

3) Удалим директорию клона:

```
postgres@tantor:~$ rm -rf $HOME/backup
```

Часть 5. Согласованная резервная копия

1) Создадим снова бэкап и расположим директорию табличного пространства `u01` вне основной директории:

```
pg_basebackup -D $HOME/backup/1 -T $PGDATA/./u01=$HOME/backup/u01 -P -c fast
4472018/4472018 kB (100%), 2/2 tablespaces
```

2) Создадим файл `standby.signal`. Если этот файл присутствует (содержимое файла не важно), что экземпляр, видя его, не открывает кластер на чтение-запись (переходит в "режим реплики"):

```
postgres@tantor:~$ touch $HOME/backup/1/standby.signal
```

Установим параметр, чтобы диагностические сообщения выводились в консоль:

```
postgres@tantor:~$ echo "logging_collector = off" >>
$HOME/backup/1/postgresql.auto.conf
```

3) Запустим экземпляр, чтобы получить "согласованную" копию. Так как бэкап автономный, то он содержит файлы журналов, нужные для согласования файлов бэкапа.

Можно запускать экземпляр командой:

```
pg_ctl start -D $HOME/backup/1 -o "--port=5433 --recovery_target=immediate --
recovery_target_action=shutdown"
```

Так как после запуска экземпляра он, достигнув согласованности, должен погаснуть (`recovery_target_action=shutdown`), то можно запустить напрямую основной процесс экземпляра. Если бы экземпляр сам не останавливался, то лучше использовать `pg_ctl`, так как надо было бы знать какой сигнал можно передать процессу `postgres`, чтобы его корректно остановить. Запустим экземпляр:

```
postgres@tantor:~$ postgres -D $HOME/backup/1 --port=5433 --
recovery_target=immediate --recovery_target_action=shutdown
```

```
LOG: starting PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
LOG: listening on IPv4 address "0.0.0.0", port 5433
LOG: listening on IPv6 address ":::", port 5433
LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5433"
LOG: database system was interrupted; last known up at
WARNING: specified neither primary_conninfo nor restore_command
HINT: The database server will regularly poll the pg_wal subdirectory to check for files placed there.
LOG: entering standby mode
LOG: redo starts at 115/BB000028
LOG: consistent recovery state reached at 115/BB000178
LOG: database system is ready to accept read-only connections
LOG: recovery stopping after reaching consistency
LOG: shutdown at recovery target
LOG: shutting down
LOG: database system is shut down
```

4) Проверим бэкап:

```
postgres@tantor:~$ pg_verifybackup $HOME/backup/1
pg_verifybackup: error: "pg_stat/pg_stat_statements.stat" is present on disk but not in the
manifest
pg_verifybackup: error: "pg_stat/pgstat.stat" is present on disk but not in the manifest
pg_verifybackup: error: "postmaster.opts" is present on disk but not in the manifest
pg_verifybackup: error: "global/pg_store_plans.stat" is present on disk but not in the manifest
pg_verifybackup: error: "backup_label.old" is present on disk but not in the manifest
pg_verifybackup: error: "backup_label" is present in the manifest but not on disk
```

При этой проверке ошибок, связанных с файлами в директории u01, нет. **Файл backup_label был переименован**, а это значит, что при использовании этого бэкапа восстановление начнется с LSN, указанных в файле pg_control.

5) Проверим LSN-записи в управляющем файле:

```
postgres@tantor:~$ pg_controldata -D $HOME/backup/1
```

```
Database cluster state:                shut down in recovery
Latest checkpoint location:            115/BB000070
Latest checkpoint's REDO location:     115/BB000028
Latest checkpoint's REDO WAL file:    0000000100000115000000BB
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:           35739
Latest checkpoint's NextOID:          399126
Latest checkpoint's NextMultiXactId:   502936
Latest checkpoint's NextMultiOffset:   2034077
Latest checkpoint's oldestXID:        723
Latest checkpoint's oldestXID's DB:    1
Latest checkpoint's oldestActiveXID:   35739
Latest checkpoint's oldestMultiXid:    1
Latest checkpoint's oldestMulti's DB:  1
Latest checkpoint's oldestCommitTsXid: 0
Latest checkpoint's newestCommitTsXid: 0
...
Fake LSN counter for unlogged rels:    0/3E8
Minimum recovery ending location:      115/BB000178
Min recovery ending loc's timeline:    1
Backup start location:                 0/0
Backup end location:                   0/0
End-of-backup record required:         no
wal_level setting:                     replica
```

Если удалить журнальный файл **0000000100000115000000BB**, то экземпляр не запустится.

С этого бэкапа нельзя (ни до согласования, ни после) восстановиться на момент раньше "**Minimum recovery ending location**".

Часть 6. Удаление файлов журнала

1) Удалите файл standby.signal:

```
postgres@tantor:~$ rm $HOME/backup/1/standby.signal
```

2) Запустите экземпляр:

```
postgres@tantor:~$ pg_ctl start -D $HOME/backup/1 -o "--port=5433"
```

```
LOG: database system was not properly shut down; automatic recovery in progress
LOG: redo starts at 115/BB000028
LOG: redo done at 115/BB000178 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed:
0.00 s
LOG: checkpoint starting: end-of-recovery immediate wait
LOG: checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1
recycled; write=0.003 s, sync=0.001 s, total=0.008 s; sync files=3, longest=0.001 s,
average=0.001 s; distance=16384 kB, estimate=16384 kB; lsn=115/BC000028, redo
lsn=115/BC000028
LOG: database system is ready to accept connections
done
server started
```

3) Корректно остановите экземпляр:

```
postgres@tantor:~$ pg_ctl stop -D $HOME/backup/1
```

```
LOG: received fast shutdown request
waiting for server to shut down....
LOG: aborting any active transactions
LOG: background worker "logical replication launcher" (PID 4137) exited with exit code 1
LOG: shutting down
LOG: checkpoint starting: shutdown immediate
LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.001 s, sync=0.001 s, total=0.007 s; sync files=0, longest=0.000 s,
average=0.000 s; distance=0 kB, estimate=14745 kB; lsn=115/BC000108, redo
lsn=115/BC000108
LOG: database system is shut down
done
server stopped
```

4) Посмотрим, что поменялось в управляющем файле после обычной остановки, по сравнению с запуском в режиме реплики:

```
postgres@tantor:~$ pg_controldata -D $HOME/backup/1
Database cluster state:          shut down
pg_control last modified:       03:58:27 AM MSK
Latest checkpoint location:     115/BC000108
Latest checkpoint's REDO location: 115/BC000108
Latest checkpoint's REDO WAL file: 0000000100000115000000BC
Latest checkpoint's TimeLineID: 1
Latest checkpoint's PrevTimeLineID: 1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:     35739
Latest checkpoint's NextOID:     399126
Latest checkpoint's NextMultiXactId: 502936
Latest checkpoint's NextMultiOffset: 2034077
Latest checkpoint's oldestXID:   723
Latest checkpoint's oldestXID's DB: 1
Latest checkpoint's oldestActiveXID: 0
Latest checkpoint's oldestMultiXid: 1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid: 0
Latest checkpoint's newestCommitTsXid: 0
```

```

Time of latest checkpoint:          03:58:27 AM MSK
Fake LSN counter for unlogged rels: 0/3E8
Minimum recovery ending location:   0/0
Min recovery ending loc's timeline: 0
Backup start location:              0/0
Backup end location:                0/0
End-of-backup record required:      no
    
```

5) Удалите **все** файлы (**0000000100000115000000BC**) журнала:

```
postgres@tantor:~$ rm -r $HOME/backup/1/pg_wal/*
```

6) Попробуйте запустить экземпляр:

```
postgres@tantor:~$ pg_ctl start -D $HOME/backup/1 -o "--port=5433"
```

```

waiting for server to start.....
LOG:  database system was shut down at 03:58:27 MSK
LOG:  creating missing WAL directory "pg_wal/archive_status"
LOG:  invalid checkpoint record
PANIC: could not locate a valid checkpoint record
LOG:  startup process (PID 4151) was terminated by signal 6: Aborted
LOG:  aborting startup due to startup process failure
LOG:  database system is shut down
      stopped waiting
pg_ctl: could not start server
Examine the log output.
    
```

Без файла журнала Latest checkpoint's REDO WAL file **0000000100000115000000BC**

экземпляр не запустился.

Вручную удалять файлы в директории pg_wal нельзя.

Как минимум один из файлов (текущий сегмент журнала) понадобится при запуске экземпляра.

7) Удалите директорию бэкапа:

```
postgres@tantor:~$ rm -rf $HOME/backup
```

Часть 7. Создание архива журнала утилитой `pg_receivewal`

1) Откройте новое окно терминала для пользователя `postgres`.

Создайте директорию для архива журналов:

```
postgres@tantor:~$ mkdir $HOME/archivelog
```

2) Запустите `pg_receivewal`:

```
postgres@tantor:~$ pg_receivewal -D $HOME/archivelog --slot=arch --synchronous -v
pg_receivewal: error: replication slot "arch" does not exist
pg_receivewal: disconnected; waiting 5 seconds to try again
pg_receivewal: error: replication slot "arch" does not exist
pg_receivewal: disconnected; waiting 5 seconds to try again
```

Будут выдаваться сообщения, что слота нет. Дальше будем изучать, как будут меняться сообщения, когда создадим слот.

3) Во втором окне терминал под пользователем `postgres` создайте бэкап с созданием и использованием слота:

```
postgres@tantor:~$ pg_basebackup -D $HOME/backup/1 -T
$PGDATA/./u01=$HOME/backup/u01 -P -C --slot=arch
```

```
4472018/4472018 kB (100%), 2/2 tablespaces
```

4) Пока бэкап делается, в окне с запущенной утилитой `pg_receivewal` будут выдаваться ошибки о том, что слот используется:

```
pg_receivewal: starting log streaming at 115/BD000000 (timeline 1)
pg_receivewal: error: could not send replication command "START_REPLICATION":
ERROR: replication slot "arch" is active for PID 5013
pg_receivewal: disconnected; waiting 5 seconds to try again
```

Один слот может использовать только одна репликационная сессия.

Мы запустили `pg_receivewal` заранее, но стоило запустить и после резервирования, пропуска журналов бы не было. Запускать утилиту заранее не обязательно. После того, как `pg_basebackup` отсоединился от экземпляра, в пределах 5 секунд подсоединился `pg_receivewal`:

```
pg_receivewal: starting log streaming at 115/BD000000 (timeline 1)
pg_receivewal: finished segment at 115/BE000000 (timeline 1)
```

`pg_receivewal` получил журнальный файл **BD**.

5) Проверим, с какого файла журнала начнется восстановление:

```
postgres@tantor:~$ cat $HOME/backup/1/backup_label
START WAL LOCATION: 115/BD000028 (file 0000000100000115000000BD)
CHECKPOINT LOCATION: 115/BD000070
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 07:48:25 MSK
LABEL: pg_basebackup base backup
START TIMELINE: 1
```

Восстановление начнется с журнала **BD**.

6) Посмотрим, какой файл текущий:

```
postgres@tantor:~$ psql
postgres=# select pg_walfile_name_offset(pg_current_wal_lsn()),
pg_current_wal_lsn();
 pg_walfile_name_offset | pg_current_wal_lsn
-----+-----
 (0000000100000115000000BE,112) | 115/BE000070
(1 строка)
```

Текущий файл **BE**.

7) Посмотрим, что получает pg_receivewal:

```
postgres@tantor:~$ ls -al $HOME/archivelog
total 32776
drwxr-xr-x  2 postgres postgres   4096 07:48 .
drwxr-xr-x 10 postgres postgres   4096 08:04 ..
-rw-r-----  1 postgres postgres 16777216 07:48 0000000100000115000000BD
-rw-r-----  1 postgres postgres 16777216 07:48 0000000100000115000000BE.partial
```

Он в настоящее время получает журнальные записи и пишет в файл **BE**. У файла расширение **.partial**. Запись идёт синхронно (поблочно: wal_block_size=8Кб), так как мы указали параметр `--synchronous`.

8) Проверим, что файл **.partial** и текущий журнал кластера одинаковы:

```
postgres@tantor:~$ diff $HOME/archivelog/0000000100000115000000BE.partial
$PGDATA/pg_wal/0000000100000115000000BE
```

Разницы нет, файлы одинаковы.

9) Посмотрим статус слота репликации:

```
postgres=# select * from pg_replication_slots \gx
-[ RECORD 1 ]-----+-----
slot_name          | arch
plugin             |
slot_type          | physical
datoid             |
database           |
temporary          | f
active             | t
active_pid         | 5018
xmin               |
catalog_xmin       |
restart_lsn        | 115/BE000198
confirmed_flush_lsn |
wal_status         | reserved
safe_wal_size      | 150994536
two_phase          | f
conflicting        |
```

```
postgres=# select * from pg_stat_replication \gx
-[ RECORD 1 ]-----+-----
pid                | 5018
usesysid           | 10
username           | postgres
application_name   | pg_receivewal
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 07:48:43.452192+03
```

```

backend_xmin |
state       | streaming
sent_lsn     | 115/BE000198
write_lsn    | 115/BE000198
flush_lsn    | 115/BE000198
replay_lsn   |
write_lag    | 00:00:00.001285
flush_lag    | 00:00:00.001285
replay_lag   | 00:27:34.008699
sync_priority | 0
sync_state | async
reply_time   | 08:16:17.463519+03
    
```

Часть 8. Синхронная фиксация транзакций и `pg_receivewal`

1) Укажем имя приложения в списке тех клиентов, которые могут подтверждать транзакции в синхронном режиме:

```
postgres=# alter system set synchronous_standby_names = pg_receivewal;
ALTER SYSTEM
postgres=# select pg_reload_conf();
 pg_reload_conf
-----
t
```

2) Обязательно убедитесь, что статус стал **sync**:

```
postgres=# select * from pg_stat_replication \gx
-[ RECORD 1 ]-----+-----
pid                | 5169
usesysid           | 10
username           | postgres
application_name   | pg_receivewal
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 08:30:00.356885+03
backend_xmin       |
state              | streaming
sent_lsn           | 115/BE000F70
write_lsn          | 115/BE000F70
flush_lsn          | 115/BE000F70
replay_lsn         |
write_lag          | 00:00:00.003395
flush_lag          | 00:00:00.003395
replay_lag         | 00:01:32.059937
sync_priority      | 1
sync_state       | sync
reply_time        | 08:31:32.419514+03
```

3) Если нет ни одного клиента со статусом **sync**, `synchronous_standby_names` не пуст, `synchronous_commit` не установлен в `local` или `off`, то транзакции будут подвисать, и при прерывании `<ctrl+c>` выдавать ошибки вида:

```
postgres=# insert into t (t) values ('aaa');
^Ccancel request sent
WARNING: canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been
replicated to the standby.
INSERT 0 1
```

Висеть сессии будут, пока не появится клиент, который подтвердит транзакции, или администратор не отключит параметрами режим.

4) Уберите режим:

```
postgres=# alter system RESET synchronous_standby_names;
ALTER SYSTEM
postgres=# select pg_reload_conf();
 pg_reload_conf
-----
t
```

Часть 9. Минимизация потерь данных транзакций

Для исключения потерь данных транзакций, нужно до потери использовать режим синхронной фиксации.

Если позволяет размер директории `pg_wal`, скопируем архивные файлы, которые понадобятся для восстановления резервной копии в директорию `pg_wal`. Файлы журнала, с которого начнется накат, указаны в `backup_label` или, если его нет (переименован в `backup_label.old`), то в `pg_control` (который просматривается `pg_controldata`). Если размер директории не позволяет, и использовать линки на уровне файловой системы не хочется, то можно использовать параметр `rectore_command`, но она будет копировать файлы журналов из архивной директории в `pg_wal`, на что тратится время и увеличивается время восстановления.

Считаем, что основной кластер у нас сбойнул и исчез. Благодаря синхронному режиму `pg_receivewal` принял все блоки текущего журнала. Если он использовался для подтверждения транзакций, то по журнальным записям (о фиксации транзакций), которые он не получил и не успел подтвердить, и клиенты выполнявшие эти транзакции не получили подтверждения о фиксации, а получили сообщение о разрыве соединения (кластер сбойнул и исчез).

Не будем тратить время на создание сессий, выдачу команд, отслеживание LSN, чтобы не отвлекаться, и сосредоточимся на основном.

1) Скопируем содержимое директории:

```
postgres@tantor:~$ cp $HOME/archivelog/* $HOME/backup/1/pg_wal
```

2) Переименуем файл `.partial`, убрав расширение:

```
postgres@tantor:~$ mv $HOME/backup/1/pg_wal/0000000100000115000000BE.partial
$HOME/backup/1/pg_wal/0000000100000115000000BE
```

3) Запустим резервный кластер:

```
postgres@tantor:~$ pg_ctl start -D $HOME/backup/1 -o "--port=5433"
```

```
LOG:  consistent recovery state reached at 115/BD000178
LOG:  invalid record length at 115/BE000F70: expected at least 26, got 0
LOG:  database system is ready to accept connections
```

что соответствует значению `sent_lsn = 115/BE000F70`, которое мы видели в `pg_stat_replication`.

4) Как останавливать `pg_receivewal`? Если не отправили в фон, как в нашем случае, то набрать в его окне **Ctrl+c**. Если отправили в фон, то можно найти номер процесса и послать сигнал `SIGINT`. Это корректное завершение `pg_receivewal`. Пример:

```
postgres@tantor$ kill -s SIGINT 5169
```

Утилита сообщит в `stdout`:

```
pg_receivewal: not renaming "0000000100000115000000BE.partial", segment is not
complete
```

5) Остановим запасной кластер:

```
postgres@tantor:~$ pg_ctl stop -D $HOME/backup/1
```

Глава 7b. Логическое резервирование

Часть 1. Использование утилиты `pg_dump`

1) Выполните команду:

```
postgres@tantor:~$ pg_dump --schema-only
```

Параметр `--schema-only` позволяет выгружать только определения объектов ("схемы объектов") без данных. Другие параметры `pg_dump` не использовали, значит использовались параметры по умолчанию:

подсоединение к базе данных, к которой подсоединился бы `psql`;

вывод в `stdout` - на экран терминала;

формат формируемого дампа `plain` - текстовый скрипт.

2) Нажимая на клавиатуре комбинацию клавиш `<Shift+PgUp>`, посмотрите как выглядит содержимое "дампа". Формат называется `plain`. "Дамп" содержит комментарии, команды `SET`, устанавливающие параметры сессии, позволяющие не зависеть от значений параметров той базы данных, в которой выполнялись бы команды из дампа:

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
```

Параметры таймаутов позволяют загружать большие объемы данных, не сталкиваясь с ограничениями.

Параметр `row_security` позволит получать ошибки в случае, если сработает политика `row level security` ("RLS"). По умолчанию, `pg_dump` откажется выгружать данные, если у роли нет прав обходить эти политики. Право обходить политики даёт атрибут роли `BYPASSRLS` и `SUPERUSER`. Это нужно, чтобы убедиться, что все строки выгружены и будут загружены без ошибок.

Параметр `check_function_bodies` отключает проверку тела подпрограмм в момент создания. Эта проверка нужна разработчикам, чтобы они видели ошибки при создании. Утилита отключает эту проверку, чтобы не заботиться о порядке выгрузки и порядке создания объектов. Это даёт гибкость: иметь возможность создать подпрограммы до создания таблиц, функций и других объектов, от которых зависят подпрограммы.

3) Создайте базу данных с названием `dump` и таблицу в этой базе данных:

```
postgres=# CREATE DATABASE dump;
CREATE DATABASE
postgres=# \c dump
Вы подключены к базе данных "dump" как пользователь "postgres".
dump=# CREATE TABLE t (id bigserial, t text, b bytea);
```

```
CREATE TABLE
dump=# INSERT INTO t(t) values ('абвг'), (NULL), ('');
INSERT 0 3
```

3) В командной строке (в другом окне терминала или выйдя из `psql`) создайте базу данных

`dump1` и перезагрузите в неё содержимое базы данных `dump`:

```
dump=# \q
postgres@tantor:~$ createdb dump1;
postgres@tantor:~$ pg_dump -d dump | psql -d dump1
...
  set_config
-----
(1 row)
...
CREATE TABLE
ALTER TABLE
CREATE SEQUENCE
ALTER SEQUENCE
ALTER SEQUENCE
ALTER TABLE
COPY 3
  setval
-----
      3
(1 строка)
```

В процессе работы `psql` выдаёт **сообщения** на экран терминала.

Утилита `pg_dump` подсоединилась к базе данных `dump` и через пайп ("`|`") передавала команды утилите `psql`, которая их тут же выполняла, подсоединившись к базе данных `dump1`.

Преимущества использования пайпа ("конвейер"):

- 1) не нужно место для файла, в который выгружались бы данные;
- 2) уменьшается время, так как одновременно работают процессы выгрузки (`pg_dump`) и загрузки (`psql`).

Часть 2. Формат custom и утилита pg_restore

1) Запустите перегрузку данных из базы `dump1` в базу `dump` в формате `custom` и

предварительным удалением объектов перед их созданием:

```
postgres@tantor:~$
pg_dump -d dump1 --format=custom | pg_restore -d dump --clean --if-exists
```

Ошибок нет. Формат `custom` формирует один файл, который может загружать не `psql`, а утилита `pg_restore`.

2) Повторите перегрузку, добавив параметр `--verbose`, чтобы посмотреть, какую информацию выдаёт:

```
postgres@tantor:~$
pg_dump -d dump1 --format=custom | pg_restore -d dump --clean --if-exists -v
```

```
pg_restore: подключение к базе данных для восстановления
pg_restore: удаляется DEFAULT t id
pg_restore: удаляется SEQUENCE t_id_seq
pg_restore: удаляется TABLE t
pg_restore: создаётся TABLE "public.t"
pg_restore: создаётся SEQUENCE "public.t_id_seq"
pg_restore: создаётся SEQUENCE OWNED BY "public.t_id_seq"
pg_restore: создаётся DEFAULT "public.t id"
pg_restore: обрабатываются данные таблицы "public.t"
pg_restore: выполняется SEQUENCE SET t_id_seq
```

3) Проверьте, что содержимое таблицы `t` соответствует исходному:

```
postgres@tantor:~$ psql -d dump -c "select * from t"
 id | t   | b
----+-----+---
  4 | абвг |
  5 |     |
  6 |     |
(3 строки)
```

4) Запустите выгрузку и загрузку утилитой `pg_restore` с параметром `--list`:

```
postgres@tantor:~$ pg_dump -d dump1 --format=custom | pg_restore -l
;
; Archive created at ..
;   dbname: dump1
;   TOC Entries: 10
;   Compression: gzip
;   Dump Version: 1.15-0
;   Format: CUSTOM
;   Integer: 4 bytes
;   Offset: 8 bytes
;   Dumped from database version: 17.5
;   Dumped by pg_dump version: 17.5
;
;
; Selected TOC Entries:
;
216; 1259 448357 TABLE public t postgres
217; 1259 448362 SEQUENCE public t_id_seq postgres
3288; 0 0 SEQUENCE OWNED BY public t_id_seq postgres
3135; 2604 448363 DEFAULT public t id postgres
3280; 0 448357 TABLE DATA public t postgres
```

```
3289; 0 0 SEQUENCE SET public t_id_seq postgres
```

Утилита `pg_restore` выдала содержимое (TOC, title of contents) дампа.

Параметр `-l` работает с дампами в формате `custom` или `directory`. Посмотрите, как выглядит список. По каждому объекту выведена строка. Строки можно закомментировать и, используя параметр `-I` утилиты `pg_restore`, не загружать эти объекты.

5) У объектов могут быть зависимости от наличия других объектов. Зависимости выводятся в списке параметром при запуске `pg_restore` с параметром `-v`. Используйте этот параметр, чтобы посмотреть как отображаются **ЗАВИСИМОСТИ**:

```
postgres@tantor:~$ pg_dump -d dump1 --format=custom | pg_restore -l -v
;
; Archive created at 2024-03-23 08:36:49 MSK
;   dbname: dump1
;   TOC Entries: 10
;   Compression: gzip
;   Dump Version: 1.15-0
;   Format: CUSTOM
;   Integer: 4 bytes
;   Offset: 8 bytes
;   Dumped from database version: 17.5
;   Dumped by pg_dump version: 17.5
;
;
; Selected TOC Entries:
;
3284; 0 0 ENCODING - ENCODING
3285; 0 0 STDSTRINGS - STDSTRINGS
3286; 0 0 SEARCHPATH - SEARCHPATH
3287; 1262 448356 DATABASE - dump1 postgres
216; 1259 448357 TABLE public t postgres
217; 1259 448362 SEQUENCE public t_id_seq postgres
;   depends on: 216
3288; 0 0 SEQUENCE OWNED BY public t_id_seq postgres
;   depends on: 217
3135; 2604 448363 DEFAULT public t id postgres
;   depends on: 217 216
3280; 0 448357 TABLE DATA public t postgres
;   depends on: 216
3289; 0 0 SEQUENCE SET public t_id_seq postgres
;   depends on: 217
```

Строки с отображением зависимостей закомментированы.

6) Если не указать утилите `pg_restore` параметры `-d` или `-l`, а указать только `-f`, то из дампа в формате `custom`, `directory`, `tar` создаётся скрипт с командами SQL. Создайте **скрипт**:

```
postgres@tantor:~$
pg_dump -d dump1 --format=custom | pg_restore -f script.sql
```

7) Создайте **скрипт** дампа в формате `plain`:

```
postgres@tantor:~$ pg_dump -d dump1 -f script1.sql
```

8) Сравните два скрипта:

```
postgres@tantor:~$ diff script.sql script1.sql
```

Скрипты не отличаются друг от друга. Утилита `pg_restore` может формировать файл дампа формата `plain` из дампов форматов `custom`, `directory`, `tar`.

Часть 3. Формат directory

1) Создайте дамп в формате `directory`:

```
postgres@tantor:~$ pg_dump -d dump1 --format=directory -f ./1
postgres@tantor:~$ ls ./1
```

```
3280.dat.gz  toc.dat
```

2) Директория создаётся автоматически. В директории лежит бинарный файл дампа и файлы с данными, для которых **по умолчанию используется сжатие**:

3) Удалите директорию и создайте дамп **без сжатия**:

```
postgres@tantor:~$ rm -rf ./1
postgres@tantor:~$ pg_dump -d dump1 --format=directory -z0 -f ./1
postgres@tantor:~$ ls ./1
3280.dat  toc.dat
```

4) Посмотрите содержимое любого файла `.dat`:

```
postgres@tantor:~$ cat ./1/3280.dat
4      абвг  \N
5      \N    \N
6      \N
\.
```

Файл `.dat` содержит результат выполнения команды `COPY` в формате по умолчанию для этой команды. `\N` - пустые (NULL) значения. `\.` - символы завершения команды `COPY`.

5) Можно выгрузить только данные, **без команд создания объектов**:

```
postgres@tantor:~$ pg_dump -d dump -a
```

в дампе будут отсутствовать команды `CREATE`.

6) Параметр `--quote-all-identifiers` указывает брать в кавычки все идентификаторы:

```
postgres@tantor:~$ pg_dump -d dump --quote-all-identifiers | grep \"

-- Name: SCHEMA "public"; Type: COMMENT; Schema: -; Owner: pg_database_owner
COMMENT ON SCHEMA "public" IS 'standard public schema';
SET default_table_access_method = "heap";
CREATE TABLE "public"."t" (
  "id" bigint NOT NULL,
  "t" "text",
  "b" "bytea"
ALTER TABLE "public"."t" OWNER TO "postgres";
CREATE SEQUENCE "public"."t_id_seq"
ALTER SEQUENCE "public"."t_id_seq" OWNER TO "postgres";
ALTER SEQUENCE "public"."t_id_seq" OWNED BY "public"."t"."id";
ALTER TABLE ONLY "public"."t" ALTER COLUMN "id" SET DEFAULT
"nextval('"public"."t_id_seq"'::regclass);
COPY "public"."t" ("id", "t", "b") FROM stdin;
SELECT pg_catalog.setval('"public"."t_id_seq"', 6, true);
```

7) Для формирования команд `INSERT` вместо команды `COPY` используется параметр

`--rows-per-insert`:

```
postgres@tantor:~$ pg_dump -d dump --rows-per-insert=1 | grep INS
INSERT INTO public.t VALUES (4, 'абвг', NULL);
INSERT INTO public.t VALUES (5, NULL, NULL);
INSERT INTO public.t VALUES (6, '', NULL);
```

Часть 4. Сжатие и скорость резервирования

1) Запустите `psql` и подключитесь к базе данных `dump`:

```
postgres@tantor:~$ psql -d dump
psql (17.5)
Введите "help", чтобы получить справку.
```

2) Выполните в `psql` следующие команды создания таблицы и наполнения её данными:

```
DROP TABLE IF EXISTS t;
CREATE TABLE t (id bigserial, t text);
INSERT INTO t(t) SELECT encode((floor(random()*1000)::numeric ^
100::numeric)::text::bytea, 'base64') from generate_series(1,500000);
```

3) Выполните следующие команды, с помощью которых можно измерить время выгрузки при использовании различных алгоритмов сжатия:

```
postgres@tantor:~$ date +%T ; rm -rf ./1 ; pg_dump -d dump --format=directory -Z
lz4 -f ./1 ; date +%T ; ls -l ./1
23:28:54
23:28:55
total 114804
-rw-r--r-- 1 postgres postgres 117547931 23:28 3281.dat.lz4
-rw-r--r-- 1 postgres postgres 2127 23:28 toc.dat
```

```
postgres@tantor:~$ date +%T ; rm -rf ./1 ; pg_dump -d dump --format=directory -Z
zstd -f ./1 ; date +%T ; ls -l ./1
23:29:17
23:29:18
total 7504
-rw-r--r-- 1 postgres postgres 7677214 23:29 3281.dat.zst
-rw-r--r-- 1 postgres postgres 2127 23:29 toc.dat
```

```
postgres@tantor:~$ date +%T ; rm -rf ./1 ; pg_dump -d dump --format=directory -Z
gzip -f ./1 ; date +%T ; ls -l ./1
23:29:31
23:29:46
total 66436
-rw-r--r-- 1 postgres postgres 68022603 23:29 3281.dat.gz
-rw-r--r-- 1 postgres postgres 2127 23:29 toc.dat
```

```
postgres@tantor:~$ date +%T ; rm -rf ./1 ; pg_dump -d dump --format=directory -Z
0 -f ./1 ; date +%T ; ls -l ./1
23:29:52
23:29:53
total 175624
-rw-r--r-- 1 postgres postgres 179830026 23:29 3281.dat
-rw-r--r-- 1 postgres postgres 2127 23:29 toc.dat
```

По результатам команд можно оценить время выгрузки в зависимости от выбранного алгоритма сжатия. Также можно менять степень сжатия, указав после названия алгоритма двоеточие и число: `-Z zstd:1`

Часть 5. Команда COPY

1) Результат команды COPY можно передавать на вход программе, например gzip:

```
dump=# COPY pg_authid TO PROGRAM 'gzip > file.gz';
COPY 17
```

В этом примере вызывается программа gzip и создаёт файл \$PGDATA/file.gz, в котором содержится текстовый файл с названием "file".

2) Можно сохранять результаты выполнения любых команд, возвращающих данные.

Например, команды WITH:

```
COPY (WITH RECURSIVE t(n) AS ( SELECT 1 UNION ALL SELECT n+1 FROM t )
SELECT n FROM t LIMIT 1
) TO stdout;
```

3) Выполните команды:

```
drop table if exists t2;
create table t2 (c1 text);
insert into t2 (c1) VALUES (repeat(E'a\n', 357913941));
COPY t2 TO '/tmp/test';
```

При выполнении последней команды выдастся ошибка:

```
ERROR: out of memory
ПОДРОБНОСТИ: Cannot enlarge string buffer containing 1073741822 bytes by 1 more bytes.
```

Размер поля - треть гигабайта.

При выгрузке в текстовом виде, содержимое поля будет выглядеть так:

a\na\na\na\n и размер поля увеличится в три раза до 1073741823 байт, что на 1 байт

превышает максимальный размер буфера строк.

4) Поле можно выгрузить, используя параметр конфигурации Tantor Postgres

enable_large_allocations = on или формат binary:

```
postgres=# COPY t2 TO '/tmp/test' WITH BINARY;
COPY 1
```

5) Удалите файл:

```
postgres=# \! rm /tmp/test
```

6) Сравним формат по умолчанию и CSV. Выполните команды:

```
postgres=# copy t to stdout with (format text);
1      абвг      \N
2      \N        \N
3      \N
postgres=# copy t to stdout with (format csv);
1,абвг,
2,,
3,"",
```

В формате CSV пустая строка была взята в кавычки.

Глава 8а. Физическая репликация

Часть 1. Создание реплики

1) Проверьте, есть ли табличные пространства:

```
postgres=# \db
      List of tablespaces
  Name      | Owner   | Location
-----+-----+-----
 pg_default | postgres |
 pg_global  | postgres |
 u01tbs    | postgres | /var/lib/postgresql/tantor-se-17/data/./u01
(3 rows)
```

2) Если есть табличные пространства, кроме двух стандартных (`pg_global`, `pg_default`), посмотрите какие отношения есть в них:

```
SELECT n.nspname, relname
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace,
pg_tablespace t
WHERE relkind IN ('r','m','i','S','t') AND
n.nspname <> 'pg_toast' AND t.oid = reltablespace AND
t.spcname = 'u01tbs';
```

```
nspname | relname
-----+-----
 public | t
(1 строка)
```

3) Удалите объекты, которые используют эти табличные пространства:

```
postgres=# drop table t;
DROP TABLE
```

4) Удалите табличное пространство `u01tbs`:

```
postgres=# drop tablespace u01tbs;
DROP TABLESPACE
```

5) Если нет второго окна терминала (`fly-term`), то откройте второе окно терминала и переключитесь в пользователя `postgres`:

```
astra@tantor:~$ su - postgres
Password: postgres
postgres@tantor:~$
```

6) Удалите директорию:

```
postgres@tantor:~$ rm -rf /var/lib/postgresql/tantor-se-17-replica/data1
```

7) Сделайте бэкап с параметрами:

- P - показывает прогресс резервирования;
- C или --slot - создает слот;
- R - создает файлы конфигурации для реплики:

```
postgres@tantor:~$ pg_basebackup -D /var/lib/postgresql/tantor-se-17-
replica/data1 -P -R -C --slot=replica1
```

Если резервирование прервать, то нужно будет удалить директорию:

```
rm -rf /var/lib/postgresql/tantor-se-17-replica/data1
```

и слот на мастере:

```
select pg_drop_replication_slot('replica1');
```

8) После успешного создания бэкапа нужно установить порт для экземпляра реплики.

Обязательно указать две угловые скобки, если будет одна, то файл затрётся:

```
echo "port=5433" >> /var/lib/postgresql/tantor-se-17-
replica/data1/postgresql.auto.conf
```

9) Чтобы диагностические сообщения выводились на экран терминала, добавьте строку в файл конфигурации:

```
echo "logging_collector = off" >> /var/lib/postgresql/tantor-se-17-
replica/data1/postgresql.auto.conf
```

Иначе диагностические сообщения будут писаться в файл директории PGDATA/**log**.

При запуске экземпляра утилитой `pg_ctl` в таком случае, будет выдаваться сообщение:

ожидание запуска сервера....

```
[pid] СООБЩЕНИЕ: передача вывода в протокол процессу сбора протоколов
```

```
[pid] ПОДСКАЗКА: В дальнейшем протоколы будут выводиться в каталог "log".
```

готово

В сообщении "ПОДСКАЗКА" выдаётся значение параметра `log_directory`. Параметр

`log_destination = stderr`, а это значит что в PGDATA создаётся файл `current_logfiles`, в который записывается расположение файлов логов, в которые пишет процесс коллектора.

10) Можно запустить реплику:

```
pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1
```

ожидание запуска сервера....

```
[7849] СООБЩЕНИЕ: запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra
12.2.0-14.astra3) 12.2.0, 64-bit
```

```
[7849] СООБЩЕНИЕ: для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5433
```

```
[7849] СООБЩЕНИЕ: для приёма подключений по адресу IPv6 ":::" открыт порт 5433
```

```
[7849] СООБЩЕНИЕ: для приёма подключений открыт Unix-сокеты "/var/run/postgresql/.s.PGSQL.5433"
```

```
[7852] СООБЩЕНИЕ: работа системы БД была прервана; последний момент работы:
```

```
[7852] СООБЩЕНИЕ: переход в режим резервного сервера
```

```
[7852] СООБЩЕНИЕ: запись REDO начинается со смещения 9/BV000028
```

```
[7852] СООБЩЕНИЕ: согласованное состояние восстановления достигнуто в позиции 9/BV000130
```

```
[7849] СООБЩЕНИЕ: система БД готова принимать подключения в режиме "только чтение"
```

```
[7853] СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени
```

1

готово

сервер запущен

Диагностические сообщения (протокол работы экземпляра) выводятся в терминал.

Реплика создана, получает без задержки журнальные записи и применяет их.

Часть 2. Слоты репликации

1) В окне терминала с `psql`, подсоединенному к мастеру, посмотрите, что слот создан и активен:

```
postgres=# select * from pg_replication_slots;
 slot_name | plugin | slot_type | datoid | database | temporary | active |
-----+-----+-----+-----+-----+-----+-----+
 replical  |       | physical  |        |          | f          | t      |
(1 строка)
```

2) Еще одно представление для мониторинга репликации:

```
postgres=# select * from pg_stat_replication \gx
-[ RECORD 1 ]-----+-----
pid          | 7854
usesysid     | 10
username     | postgres
application_name | walreceiver
client_addr  |
client_hostname |
client_port  | -1
backend_start | 13:56:31.619654+03
backend_xmin  |
state        | streaming
sent_lsn     | 9/BC000198
write_lsn    | 9/BC000198
flush_lsn    | 9/BC000198
replay_lsn   | 9/BC000198
write_lag    |
flush_lag    |
replay_lag   |
sync_priority | 0
sync_state   | async
reply_time   | 14:24:31.557301+03
```

Имя приложения по умолчанию **walreceiver**.

3) Подключитесь к реплике:

```
postgres=# \connect postgres postgres /var/run/postgresql 5433
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5433".
```

4) Посмотрите название слота репликации, к которому подсоединяется реплика:

```
postgres=# \dconfig primary_slot_name
List of configuration parameters
Parameter | Value
-----+-----
primary_slot_name | replical
(1 строка)
```

5) Посмотрите значение параметра `cluster_name`:

```
postgres=# \dconfig cluster_name
List of configuration parameters
Parameter | Value
-----+-----
cluster_name |
```

Значение параметра `cluster_name` пусто, поэтому значение параметра `application_name` имеет значение по умолчанию `walreceiver`.

6) Посмотрите значение параметра `primary_conninfo`:

```
postgres=# show primary_conninfo \gx
-[ RECORD 1 ]-----
primary_conninfo | user=postgres passfile='/var/lib/postgresql/.pgpass'
channel_binding=prefer port=5432 sslmode=prefer sslcompression=0
sslcertmode=allow sslsni=1 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer
krbsrvname=postgres gssdelegation=0 compression=off target_session_attrs=any
load_balance_hosts=disable
```

Часть 3. Изменение имени кластера

1) Установите значение параметра `cluster_name`:

```
postgres=# alter system set cluster_name = 'replica1';
ALTER SYSTEM
```

2) На реплике представление `pg_stat_replication` пусто:

```
postgres=# select * from pg_stat_replication;
 pid | usesysid | username | application_name | client_addr | client_hostname
-----+-----+-----+-----+-----+-----
(0 строк)
```

3) Изменение параметра `cluster_name` требует рестарта экземпляра. Перезапустите экземпляр реплики в окне терминала:

```
postgres@tantor:~$
pg_ctl restart -D /var/lib/postgresql/tantor-se-17-replica/data1
```

ожидание завершения работы сервера....

```
готово
сервер остановлен
ожидание запуска сервера....
[25550] СООБЩЕНИЕ: запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra
12.2.0-14.astra3) 12.2.0, 64-bit
[25550] СООБЩЕНИЕ: для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5433
[25550] СООБЩЕНИЕ: для приёма подключений по адресу IPv6 "::" открыт порт 5433
[25550] СООБЩЕНИЕ: для приёма подключений открыт Unix-сокет "/var/run/postgresql/.s.PGSQL.5433"
[25553] СООБЩЕНИЕ: система БД была выключена в процессе восстановления: 14:37:36 MSK
[25553] СООБЩЕНИЕ: переход в режим резервного сервера
[25553] СООБЩЕНИЕ: запись REDO начинается со смещения 9/BC000070
[25553] СООБЩЕНИЕ: согласованное состояние восстановления достигнуто в позиции 9/BC000198
[25553] СООБЩЕНИЕ: неверная длина записи в позиции 9/BC000198: ожидалось минимум 26, получено 0
[25550] СООБЩЕНИЕ: система БД готова принимать подключения в режиме "только чтение"
[25554] СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии
времени 1
готово
сервер запущен
```

4) Посмотрите список процессов в названии которых присутствует буквосочетание `wal`:

```
postgres@tantor:~$ ps -ef | grep wal
UID          PID    PPID  CMD
postgres    2654  13810 postgres: 11/main: walwriter
70           11476 13796 postgres: walwriter
postgres    13539 13534 postgres: walwriter
postgres    25554 25550 postgres: replica1: walreceiver
postgres    25555 13534 postgres: walsender postgres [local] streaming 9/BC000198
postgres    26488 31415 grep wal
```

В списке есть процессы:

`walsender` - передаёт журнальную запись

`walwriter` - принимает то, что ему передаёт `walsender`

PPID=**25550** - это номер родительского процесса (Parent Process ID) для процесса с PID=**25554**.

В данном примере номер процесса `postgres` мастера - **13534**, `walsender` - **25555**.

5) Посмотрите список процессов мастера:

```
postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17/data/postmaster.pid`
PID COMMAND
13535 postgres: logger
13536 postgres: checkpointer
13537 postgres: background writer
13539 postgres: walwriter
```

```
13540 postgres: autovacuum launcher
13541 postgres: logical replication launcher
25555 postgres: walsender postgres [local] streaming 9/BC000198
```

Процесс postgres, который их запустил, не выводится.

6) Посмотрите список процессов реплики:

```
postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17-replica/data/postmaster.pid`
  PID COMMAND
25551 postgres: replicat1: checkpointer
25552 postgres: replicat1: background writer
25553 postgres: replicat1: startup recovering 00000001000000090000000BC
25554 postgres: replicat1: walreceiver
```

После установки значения `cluster_name` у процессов реплики присутствует идентификатор `replicat1`.

7) Сделаем префиксирование названий процессов для мастера, подсоединимся к мастеру:

```
postgres=# \c postgres postgres /var/run/postgresql 5432
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5432".
```

8) Выполните команду:

```
postgres=# alter system set cluster_name = 'master';
ALTER SYSTEM
```

9) Изменение параметра `cluster_name` требует рестарта экземпляра, перезапустите экземпляр мастера в окне терминала:

```
postgres@tantor:~$ sudo systemctl restart tantor-se-server-17
```

В окне терминала, в котором выполнялась команда `pg_ctl start`, для запуска реплики будут выданы диагностические сообщения экземпляра реплики:

```
[25554] СООБЩЕНИЕ: репликация прекращена главным сервером
[25554] ПОДРОБНОСТИ: На линии времени 1 в 9/BC000230 достигнут конец журнала.
[25554] ВАЖНО: не удалось отправить главному серверу сообщение о конце передачи: сервер неожиданно закрыл
соединение
    Скорее всего сервер прекратил работу из-за сбоя
    до или в процессе выполнения запроса.
    операция COPY не выполняется
[25553] СООБЩЕНИЕ: неверная длина записи в позиции 9/BC000230: ожидалось минимум 26, получено 0
[5727] ВАЖНО: не удалось подключиться к главному серверу: подключиться к серверу через сокет
"/var/run/postgresql/.s.PGSQL.5432" не удалось: сервер неожиданно закрыл соединение
    Скорее всего сервер прекратил работу из-за сбоя
    до или в процессе выполнения запроса.
[25553] СООБЩЕНИЕ: waiting for WAL to become available at 9/BC00024A
[5782] СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени 1
[25551] СООБЩЕНИЕ: начата точка перезапуска: time
[25551] СООБЩЕНИЕ: точка перезапуска завершена: записано буферов: 1 (0.0%); добавлено файлов WAL 0, удалено: 0,
переработано: 0; запись=0.002 сек., синхр.=0.001 сек., всего=0.010 сек.; синхронизировано_файлов=0,
самая_долгая_синхр.=0.000 сек., средняя=0.000 сек.; расстояние=0 kB, ожидалось=0 kB; lsn=9/BC000198, lsn
redo=9/BC000198
[25551] СООБЩЕНИЕ: точка перезапуска восстановления в позиции 9/BC000198
```

10) Посмотрите список процессов реплики:

```
postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17-replica/data/postmaster.pid`
  PID COMMAND
 5782 postgres: replicat1: walreceiver streaming 9/BC0003A0
25551 postgres: replicat1: checkpointer
25552 postgres: replicat1: background writer
```

```
25553 postgres: replical: startup recovering 0000000100000009000000BC
```

Предыдущий процесс `walreceiver` 25554 был остановлен и выгружен из памяти. Был запущен процесс `walreceiver` 5725, но он не смог подсоединиться, так как экземпляр мастера отказал в соединении. Был запущен процесс `walreceiver` 5782, который успешно подсоединился к мастеру и принимает журнальные данные.

1) Посмотрите список процессов мастера:

```
postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17/data/postmaster.pid`
PID COMMAND
5743 postgres: master: logger
5751 postgres: master: checkpointer
5752 postgres: master: background writer
5755 postgres: master: walwriter
5756 postgres: master: autovacuum launcher
5757 postgres: master: logical replication launcher
5783 postgres: master: walsender postgres [local] streaming 9/BC000278
```

Теперь после имени процессов мастера указано значение параметра `cluster_name`.

Часть 4. Создание второй реплики

1) Создайте на мастере слот для второй реплики:

```
postgres=# select pg_copy_physical_replication_slot('replica1','replica2');
 pg_copy_physical_replication_slot
-----
 (replica2,)
(1 строка)
```

2) Посмотрите список слотов:

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
pgstandby1	f	0/19187E70	lost
replica1	t	9/BC0003A0	reserved
replica2	f	9/BC0003A0	reserved

(3 строки)

Второй слот будет удерживать журнальные файлы, начиная с файла, в котором содержится журнальная запись с адресом `restart_lsn`.

В списке может присутствовать слот `pgstandby1`. Это слот реплики, которая была изначально в виртуальной машине. Эту реплику и слот можно будет удалить за ненадобностью.

3) Сгенерируем журнальные записи на мастере. Выполните контрольную точку:

```
postgres=# checkpoint;
CHECKPOINT
```

в лог сообщений кластера master в директории `PGDATA/log` выведется:

```
[5751] LOG: checkpoint starting: immediate force wait
[5751] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0
removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.009 s; sync files=0,
longest=0.000 s, average=0.000 s; distance=0 kB, estimate=0 kB; lsn=9/BC0003E8,
redo lsn=9/BC0003A0
```

4) Посмотрим, как изменился `restart_lsn`. Выполните запрос к списку слотов:

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
pgstandby1	f	0/19187E70	lost
replica1	t	9/BC0004C8	reserved
replica2	f	9/BC0003A0	reserved

(3 строки)

Первая реплика получила сгенерированную журнальную запись, и значение сдвинулось. Для второго слота значение не изменилось.

5) Создадим вторую реплику. Чтобы не нагружать мастер, сделаем бэкап, копируя файлы с реплики ("backup offloading").

```
postgres@tantor:~$ pg_basebackup -p 5433 -D /var/lib/postgresql/tantor-se-17-
replica/data2 -P -R
466575/466575 КБ (100%), табличное пространство 1/1
```

В случае ошибки можно удалить бэкап:

```
rm -rf /var/lib/postgresql/tantor-se-17-replica/data2
```

6) Добавьте параметр `port=5434` и `logging_collector = off` для второй реплики.

Можно отредактировать файл текстовым редактором, можно добавить параметр в конец файла.

Последнее значение превалирует.

```
postgres@tantor:~$ echo "port=5434" >> /var/lib/postgresql/tantor-se-17-replica/data2/postgresql.auto.conf
```

```
postgres@tantor:~$ echo "logging_collector = off" >> /var/lib/postgresql/tantor-se-17-replica/data2/postgresql.auto.conf
```

7) Посмотрите содержимое файла `postgresql.auto.conf` новой реплики:

```
postgres@tantor:~/tantor-se-17-replica/data2$ cat /var/lib/postgresql/tantor-se-17-replica/data2/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_slot_wal_keep_size = '128MB'
max_wal_size = '128MB'
min_wal_size = '512MB'
idle_in_transaction_session_timeout = '100min'
primary_conninfo = 'user=postgres passfile='/var/lib/postgresql/.pgpass''
channel_binding=prefer port=5432 sslmode=prefer sslcompression=0
sslcertmode=allow sslsni=1 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer
krbsrvname=postgres gssdelegation=0 compression=off target_session_attrs=any
load_balance_hosts=disable'
primary_slot_name = 'replica1'
port = '5433'
logging_collector = 'off'
cluster_name = 'replica1'
primary_conninfo = 'user=postgres passfile='/var/lib/postgresql/.pgpass''
channel_binding=prefer port=5433 sslmode=prefer sslcompression=0
sslcertmode=allow sslsni=1 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer
krbsrvname=postgres gssdelegation=0 compression=off target_session_attrs=any
load_balance_hosts=disable'
port=5434
logging_collector = off
```

`pg_basebackup` резервировал, подключившись к первой реплике и поставил её порт 5433

в параметр `primary_conninfo`. С таким значением получается каскадирование передачи журнальных данных.

8) Отредактируйте файл `/var/lib/postgresql/tantor-se-17-`

`replica/data2/postgresql.auto.conf`, установив порт 5432: пусть вторая реплика подсоединяется к мастеру напрямую, имя слота и кластера в `replica2`.

Пример содержимого файла после редактирования:

```
postgres@tantor:~$ cat /var/lib/postgresql/tantor-se-17-replica/data2/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_slot_wal_keep_size = '128MB'
```

```

max_wal_size = '128MB'
min_wal_size = '512MB'
idle_in_transaction_session_timeout = '100min'
primary_conninfo = 'user=postgres port=5432'
primary_slot_name = 'replica2'
cluster_name = 'replica2'
port=5434
logging_collector = off
    
```

9) Запустите вторую реплику:

```
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data2
```

ожидание запуска сервера....

```

[5728] СООБЩЕНИЕ:  запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc
(Astra 12.2.0-14.astra3) 12.2.0, 64-bit
[5728] СООБЩЕНИЕ:  для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5434
[5728] СООБЩЕНИЕ:  для приёма подключений по адресу IPv6 ":::" открыт порт 5434
[5728] СООБЩЕНИЕ:  для приёма подключений открыт Unix-сокет
"/var/run/postgresql/.s.PGSQL.5434"
[5731] СООБЩЕНИЕ:  система БД была выключена в процессе восстановления:
[5731] СООБЩЕНИЕ:  переход в режим резервного сервера
[5731] СООБЩЕНИЕ:  запись REDO начинается со смещения 9/BC0003A0
[5731] СООБЩЕНИЕ:  согласованное состояние восстановления достигнуто в позиции 9/BC0004C8
[5728] СООБЩЕНИЕ:  система БД готова принимать подключения в режиме "только чтение"
[5731] СООБЩЕНИЕ:  неверная длина записи в позиции 9/BC0004C8: ожидалось минимум 26,
получено 0
[5732] СООБЩЕНИЕ:  начало передачи журнала с главного сервера, с позиции 9/BC000000 на
линии времени 1
    готово
сервер запущен
    
```

10) Проверьте статус слотов:

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
pgstandby1	f	0/19187E70	lost
replica1	t	9/BC0004C8	reserved
replica2	t	9/BC0004C8	reserved

(3 строки)

Слоты активны. Сейчас у вас мастер и две реплики, которые получают журнальные записи по протоколу репликации (поток). `restart_lsn` продвигается на обеих слотах.

11) Сгенерируем журнальные записи. Выполните контрольную точку:

```
postgres=# checkpoint;
CHECKPOINT
```

12) Повторите запрос к `pg_replication_slots`:

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
pgstandby1	f	0/19187E70	lost
replica1	t	9/BC0005F0	reserved
replica2	t	9/BC0005F0	reserved

(3 строки)

Сообщения экземпляров реплик:

```
[5729] СООБЩЕНИЕ:  начата точка перезапуска: time
```

```
[5729] СООБЩЕНИЕ: точка перезапуска завершена: записано буферов: 1 (0.0%);  
добавлено файлов WAL 0, удалено: 0, переработано: 0; запись=0.002 сек.,  
синхр.=0.001 сек., всего=0.008 сек.; синхронизировано_файлов=0,  
самая_долгая_синхр.=0.000 сек., средняя=0.000 сек.; расстояние=0 кВ, ожидалось=0  
кВ; lsn=9/BC000510, lsn redo=9/BC0004C8  
[5729] СООБЩЕНИЕ: точка перезапуска восстановления в позиции 9/BC0004C8  
  
[25551] СООБЩЕНИЕ: начата точка перезапуска: time  
[25551] СООБЩЕНИЕ: точка перезапуска завершена: записано буферов: 0 (0.0%);  
добавлено файлов WAL 0, удалено: 0, переработано: 0; запись=0.001 сек.,  
синхр.=0.001 сек., всего=0.006 сек.; синхронизировано_файлов=0,  
самая_долгая_синхр.=0.000 сек., средняя=0.000 сек.; расстояние=0 кВ, ожидалось=0  
кВ; lsn=9/BC000510, lsn redo=9/BC0004C8  
[25551] СООБЩЕНИЕ: точка перезапуска восстановления в позиции 9/BC0004C8
```

Точка перезапуска - отражение контрольной точки мастера.

Часть 5. Выбор реплики на роль мастера

Симулируем сбой получения журнальных записей одной из реплик, например **второй**.

Например, сделаем недоступной запись в журнальный файл и перезапустим экземпляр.

Перезапуск нужен, чтобы при открытии файла возникла ошибка:

```
1) postgres@tantor:~$ chmod -w /var/lib/postgresql/tantor-se-17-
replica/data2/pg_wal/000*
postgres@tantor:~$ pg_ctl restart -D /var/lib/postgresql/tantor-se-17-
replica/data2

12:19:48.996 MSK [5728] СООБЩЕНИЕ:   получен запрос на быстрое выключение
ожидание завершения работы сервера....
12:19:48.998 MSK [5728] СООБЩЕНИЕ:   прерывание всех активных транзакций
12:19:48.998 MSK [5732] ВАЖНО:   завершение процесса считывания журнала по команде администратора
12:19:49.004 MSK [5729] СООБЩЕНИЕ:   выключение
12:19:49.017 MSK [5728] СООБЩЕНИЕ:   система БД выключена
    готово
сервер остановлен
ожидание запуска сервера....
12:19:49.142 MSK [24184] СООБЩЕНИЕ:   запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled
by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
12:19:49.142 MSK [24184] СООБЩЕНИЕ:   для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт
5434
12:19:49.142 MSK [24184] СООБЩЕНИЕ:   для приёма подключений по адресу IPv6 ":::" открыт порт 5434
12:19:49.144 MSK [24184] СООБЩЕНИЕ:   для приёма подключений открыт Unix-сокет
"/var/run/postgresql/.s.PGSQL.5434"
12:19:49.149 MSK [24187] СООБЩЕНИЕ:   система БД была выключена в процессе восстановления: 12:19:48
MSK
12:19:49.149 MSK [24187] СООБЩЕНИЕ:   переход в режим резервного сервера
12:19:49.152 MSK [24187] СООБЩЕНИЕ:   запись REDO начинается со смещения 9/BC0004C8
12:19:49.152 MSK [24187] СООБЩЕНИЕ:   согласованное состояние восстановления достигнуто в позиции
9/BC0005F0
12:19:49.152 MSK [24187] СООБЩЕНИЕ:   неверная длина записи в позиции 9/BC0005F0: ожидалось минимум
26, получено 0
12:19:49.152 MSK [24184] СООБЩЕНИЕ:   система БД готова принимать подключения в режиме "только
чтение"
12:19:49.160 MSK [24188] СООБЩЕНИЕ:   начало передачи журнала с главного сервера, с позиции
9/BC000000 на линии времени 1
12:19:49.160 MSK [24188] ВАЖНО:   не удалось открыть файл "pg_wal/0000000100000009000000BC":
Отказано в доступе
12:19:49.167 MSK [24190] СООБЩЕНИЕ:   начало передачи журнала с главного сервера, с позиции
9/BC000000 на линии времени 1
12:19:49.168 MSK [24190] ВАЖНО:   не удалось открыть файл "pg_wal/0000000100000009000000BC":
Отказано в доступе
12:19:49.168 MSK [24187] СООБЩЕНИЕ:   waiting for WAL to become available at 9/BC00060A
    готово
сервер запущен
```

В журнал кластера будут выдаваться ошибки раз в **5 секунд** (значение параметра

`wal_retrieve_retry_interval`):

```
12:19:54.173 MSK [24232] СООБЩЕНИЕ:   начало передачи журнала с главного сервера,
с позиции 9/BC000000 на линии времени 1
12:19:54.174 MSK [24232] ВАЖНО:   не удалось открыть файл
"pg_wal/0000000100000009000000BC": Отказано в доступе
12:19:54.174 MSK [24187] СООБЩЕНИЕ:   waiting for WAL to become available at
9/BC00060A
```

2) Поменяйте интервал повторения попыток `walreceiver` с 5 секунд на 30 секунд. В окне

терминала `psql` подключитесь ко **второй реплике**:

```
postgres=# \c postgres postgres /var/run/postgresql 5434
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5434".
```

```
postgres=# alter system set wal_retrieve_retry_interval='30s';
ALTER SYSTEM
postgres=# select pg_reload_conf();
 pg_reload_conf
-----
t
(1 строка)
```

Ошибки у второй реплики станут выдаваться реже, раз в 30 секунд.

3) Заставим мастер создать журнальные записи. Подсоединитесь к мастеру и выполните контрольную точку:

```
postgres=# \c postgres postgres /var/run/postgresql 5432
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5432".
postgres=# checkpoint;
CHECKPOINT
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
 slot_name | active | restart_lsn | wal_status
-----+-----+-----+-----
pgstandby1 | f      | 0/19187E70 | lost
replica1   | t      | 9/BC0006D0 | reserved
replica2   | f      | 9/BC0005F0 | reserved
(3 строки)
```

Статус второй реплики неактивен и restart_lsn стал разным.

4) Симулируем сбой мастера. Остановите мастер:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data
ожидание завершения работы сервера....

12:40:35.444 MSK [5782] СООБЩЕНИЕ: репликация прекращена главным сервером
12:40:35.444 MSK [5782] ПОДРОБНОСТИ: На линии времени 1 в 9/BC0007B0 достигнут конец журнала.
12:40:35.444 MSK [5782] ВАЖНО: не удалось отправить главному серверу сообщение о конце передачи:
сервер неожиданно закрыл соединение
Скорее всего сервер прекратил работу из-за сбоя до или в процессе выполнения запроса.
операция COPY не выполняется
12:40:35.444 MSK [25553] СООБЩЕНИЕ: неверная длина записи в позиции 9/BC0007B0: ожидалось минимум
26, получено 0
12:40:35.453 MSK [753] ВАЖНО: не удалось подключиться к главному серверу: подключиться к серверу
через сокет "/var/run/postgresql/.s.PGSQL.5432" не удалось: сервер неожиданно закрыл соединение
Скорее всего сервер прекратил работу из-за сбоя до или в процессе выполнения запроса.
12:40:35.453 MSK [25553] СООБЩЕНИЕ: waiting for WAL to become available at 9/BC0007CA
готово
сервер остановлен
```

5) Устраним проблему на второй реплике. Верните разрешения на файл журнала:

```
postgres@tantor:~$ chmod +w /var/lib/postgresql/tantor-se-17-
replica/data2/pg_wal/000*
```

6) Имея две реплики в случае сбоя мастера, нужно выбрать ту реплику, которую лучше сделать мастером.

Посмотрите, какие журнальные записи имеются на [первой](#) реплике:

```
postgres@tantor:~$ \c postgres postgres /var/run/postgresql 5433
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5433".
postgres=# select pg_last_wal_replay_lsn();
 pg_last_wal_replay_lsn
-----
9/BC0007B0
```

(1 строка)

```
postgres=# select pg_last_wal_receive_lsn();
pg_last_wal_receive_lsn
-----
9/BC0007B0
(1 строка)
```

7) Посмотрите, какие журнальные записи имеются на **второй** реплике:

```
postgres@tantor:~$ \c postgres postgres /var/run/postgresql 5434
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5434".
postgres=# select pg_last_wal_replay_lsn();
pg_last_wal_replay_lsn
-----
9/BC0005F0
(1 строка)
```

```
postgres=# select pg_last_wal_receive_lsn();
pg_last_wal_receive_lsn
-----
9/BC000000
(1 строка)
```

8) При активной работе рассчитать, какое значение LSN больше, сложно.

Вычислите, подставив значения, которые взяты у реплики:

```
postgres=# select '9/BC0005F0'::pg_lsn - '9/BC0007B0'::pg_lsn;
?column?
-----
-448
(1 строка)
```

У первой реплики значения больше, значит она содержит последние изменения.

Мы не включали режим фиксации транзакций с подтверждением хотя бы одной из реплик. В реальной эксплуатации, в таком случае нет гарантий, что хоть какая-то реплика получила последние журнальные записи. В случае продвижения любой из реплик, часть транзакций может быть потеряна, что неприемлемо.

В случае, если синхронная фиксация с подтверждением не была включено, стоит поискать журнальные файлы на мастере или, если они повреждены, в поточном архиве журналов (заполняемый утилитой `pg_receivewal`), если он был настроен. При использовании файлов из архива нужно будет скопировать файл текущего журнала. Его легко идентифицировать по суффиксу `.partial` в названии. При копировании в директорию реплики, который планируется сделать мастером (чтобы реплика накатила файл,) нужно убрать суффикс.

9) Рассмотрим случай, когда `PGDATA/pg_wal` мастера была найдена. Эта директория содержит последние журнальные записи, которые сохранил мастер. Скопируем все файлы в директорию `PGDATA/pg_wal` второй реплики (она не получила последние журнальные записи от мастера):

```
postgres@tantor:~$ cp /var/lib/postgresql/tantor-se-17/data/pg_wal/*
/var/lib/postgresql/tantor-se-17-replica/data2/pg_wal
cp: не указан -r; пропускается каталог '/var/lib/postgresql/tantor-se-
17/data/pg_wal/archive_status'
```

Почему копируем все файлы? Потому что мастер сохраняет файлы журналов для того, чтобы иметь возможность восстановиться после сбоя экземпляра, и удерживает файлы для реплик.

Чтобы не тратить время на изучение, каких файлов не достаёт реплике, можно скопировать все файлы журналов. Те, которые не нужны реплике повторно применяться не будут.

10) Посмотрим, какие журнальные записи применены (процессом `startup`, который читает директорию `pg_wal` с журналами и применяет файлы из неё), и были приняты (процессом `walreceiver`, который принимает журнальные записи и пишет в файлы журналов в директории `pg_wal`) на **второй** реплике:

```
postgres=# select pg_last_wal_replay_lsn();
 pg_last_wal_replay_lsn
-----
 9/BC0007B0
(1 строка)
```

```
postgres=# select pg_last_wal_receive_lsn();
 pg_last_wal_receive_lsn
-----
 9/BC000000
(1 строка)
```

По `walreceiver` изменений нет, мастер остановлен и процесс ничего не мог принять.

Сейчас **обе реплики накатили все журнальные записи и содержат все данные. При продвижении любой из реплик потери транзакций не будет.**

Первая реплика успела получить все записи, потому что мы корректно остановили мастер в то время, когда первая реплика была к нему подсоединена. На вторую реплику мы скопировали все файлы журналов мастера.

Часть 6. Подготовка к переключению на реплику

Настроим параметры конфигурации бывшего мастера.

Имя слота в параметре `primary_slot_name` можно задать заранее. После переключения на реплику слоты исчезнут - на новом мастере их не будет.

Параметры соединения `primary_conninfo` также можно установить заранее. Значение порта укажем на первую реплику 5433, её сделаем матером.

Большая часть параметров, которые относятся к свойствам реплики, не оказывают влияние, пока у кластера роль мастера, поэтому значения таких параметров можно устанавливать заранее.

1) Посмотрите содержимое файла параметров бывшего мастера:

```
postgres@tantor:~$ cat /var/lib/postgresql/tantor-se-17/data/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_slot_wal_keep_size = '128MB'
max_wal_size = '128MB'
min_wal_size = '512MB'
idle_in_transaction_session_timeout = '100min'
cluster_name = 'master'
```

2) Установите параметры сетевого соединения, откуда бывший мастер будет забирать журнальные данные:

```
postgres@tantor:~$ echo "primary_conninfo = 'user=postgres port=5433'" >>
/var/lib/postgresql/tantor-se-17/data/postgresql.auto.conf
```

3) Установим название слота, который он будет использовать:

```
postgres@tantor:~$ echo "primary_slot_name = 'master'" >>
/var/lib/postgresql/tantor-se-17/data/postgresql.auto.conf
```

4) Чтобы диагностические сообщения выводились на экран терминала:

```
echo "logging_collector = off" >> /var/lib/postgresql/tantor-se-17/data/postgresql.auto.conf
```

5) Проверьте, что строки **добавились**:

```
postgres@tantor:~$ cat /var/lib/postgresql/tantor-se-17/data/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_slot_wal_keep_size = '128MB'
max_wal_size = '128MB'
min_wal_size = '512MB'
idle_in_transaction_session_timeout = '100min'
cluster_name = 'master'
primary_conninfo = 'user=postgres port=5433'
primary_slot_name = 'master'
logging_collector = off
```

6) Можно заранее создать неинициализированные слоты репликации на случай, если кластер станет мастером на всех репликах-кандидатах.

Создайте слот на второй реплике:

```
postgres@tantor:~$ psql -p 5434
```

```
psql (17.5)
Введите "help", чтобы получить справку.
```

```
postgres=# select pg_is_in_recovery();
 pg_is_in_recovery
-----
 t
(1 row)
```

```
postgres=# select pg_create_physical_replication_slot('master');
 pg_create_physical_replication_slot
-----
 (master,)
(1 row)
```

7) Заранее создадим слот и для первой реплики. Выполните команду:

```
postgres=# select pg_create_physical_replication_slot('replica1');
 pg_create_physical_replication_slot
-----
 (replica1,)
(1 row)
```

8) Проверьте параметры слотов:

```
postgres=# select * from pg_replication_slots \gx
-[ RECORD 1 ]-----+-----
slot_name          | master
plugin             |
slot_type          | physical
datoid             |
database           |
temporary          | f
active            | f
active_pid         |
xmin              |
catalog_xmin       |
restart_lsn        |
confirmed_flush_lsn |
wal_status         |
safe_wal_size      | 150994944
two_phase          | f
conflicting        |
-[ RECORD 2 ]-----+-----
slot_name          | replica1
plugin             |
slot_type          | physical
datoid             |
database           |
temporary          | f
active            | f
active_pid         |
xmin              |
catalog_xmin       |
restart_lsn        |
confirmed_flush_lsn |
wal_status         |
safe_wal_size      | 150994944
two_phase          | f
conflicting        |
```

На будущем мастере `replica1` (порт 5433) слоты создавать заранее не будем, в целях практики (пункт 11 этой части практики).

Слоты имеет смысл создать заранее, это уменьшит количество команд, выполняемых при переключении на реплику.

Значение `safe_wal_size=144Мб=128Мб+16Мб` определяет сколько байт может быть записано в журнал, чтобы этот слот не оказался в состоянии `lost`. Определяется значением параметра `max_slot_wal_keep_size=128Мб` плюс `wal_segment_size=16Мб`.

9) Так как бывший мастер остановлен, можно создать файл `standby.signal`, чтобы при запуске экземпляр не открыл бывший мастер в режиме записи. Создайте файл `standby.signal` в директории бывшего мастера:

```
postgres@tantor:~$ touch /var/lib/postgresql/tantor-se-17/data/standby.signal
```

После создания файла `standby.signal` можно запустить бывший мастер и потом продвинуть одну из реплик. Или в обратном порядке: продвинуть одну из реплик, а потом запустить бывший мастер. Разницы не будет, если бывший мастер был остановлен и новый мастер получил и применил все журнальные записи (нет потерь транзакций).

10) Запустите бывший мастер:

```
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17/data
ожидание запуска сервера....
СООБЩЕНИЕ: запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0,
64-bit
[7824] СООБЩЕНИЕ: для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5432
[7824] СООБЩЕНИЕ: для приёма подключений по адресу IPv6 ":::" открыт порт 5432
[7824] СООБЩЕНИЕ: для приёма подключений открыт Unix-сокеты "/var/run/postgresql/.s.PGSQL.5432"
[7827] СООБЩЕНИЕ: система БД была выключена: 12:40:35 MSK
[7827] СООБЩЕНИЕ: переход в режим резервного сервера
[7827] СООБЩЕНИЕ: согласованное состояние восстановления достигнуто в позиции 9/BC0007B0
[7827] СООБЩЕНИЕ: неверная длина записи в позиции 9/BC0007B0: ожидалось минимум 26, получено 0
[7824] СООБЩЕНИЕ: система БД готова принимать подключения в режиме "только чтение"
[7828] СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени 1
[7828] ОШИБКА: слот репликации "master" не существует
[7828] STATEMENT: START_REPLICATION SLOT "master" 9/BC000000 TIMELINE 1
[7828] ВАЖНО: не удалось начать трансляцию WAL: ОШИБКА: слот репликации "master" не существует
[7828] СООБЩЕНИЕ: waiting for WAL to become available at 9/BC0007CA
готово
сервер запущен
```

Экземпляр бывшего мастера запустился и успешно вошел в режим ожидания запуска репликации. Ошибки указывают на то, что слот с именем `master` не существует. Мы его заранее не создавали (в пункте 8 этой части практики) в целях получить эту ошибку и исправить её - создать слот после запуска бывшего мастера.

11) Подключитесь к первой реплике `replica1` (порт 5433) и создайте слоты репликации:

```
postgres=# \c postgres postgres /var/run/postgresql 5433
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5433".
postgres=# select pg_create_physical_replication_slot('master');
 pg_create_physical_replication_slot
-----
(master,)
(1 row)

postgres=# select pg_create_physical_replication_slot('replica2');
```

```
pg_create_physical_replication_slot
```

```
(replica2,)
(1 row)
```

БЫВШИЙ master автоматически станет использовать созданный слот.

В сообщениях экземпляра master появится:

СООБЩЕНИЕ: waiting for WAL to become available at 9/BC0007CA

СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени 1

12) Проверьте на replica1 (порт 5433) статус слотов репликации:

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
master	t	9/BC0007B0	reserved
replica2	f		

(2 строки)

13) Проверьте статистику слотов репликации на бывшем мастере (master):

```
postgres=# \c postgres postgres /var/run/postgresql 5432
```

Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в "/var/run/postgresql", порт "5432".

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
pgstandby1	f	0/19187E70	lost
replica1	t	9/BC0007B0	reserved
replica2	t	9/BC0007B0	reserved

(3 строки)

```
postgres=# select * from pg_stat_replication \gx
```

```
-[ RECORD 1 ]-----+-----
pid          | 18624
usesysid     | 10
username     | postgres
application_name | replica1
client_addr  |
client_hostname |
client_port  | -1
backend_start | 16:07:37.4+03
backend_xmin  |
state        | streaming
sent_lsn     | 9/BC0007B0
write_lsn    | 9/BC0007B0
flush_lsn    | 9/BC0007B0
replay_lsn   | 9/BC0007B0
write_lag    |
flush_lag    |
replay_lag   |
sync_priority | 0
sync_state   | async
reply_time   | 16:31:37.866881+03
-[ RECORD 2 ]-----+-----
pid          | 18693
usesysid     | 10
username     | postgres
```

```

application_name | replica2
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 16:07:52.385223+03
backend_xmin     |
state            | streaming
sent_lsn         | 9/BC0007B0
write_lsn        | 9/BC0007B0
flush_lsn        | 9/BC0007B0
replay_lsn       | 9/BC0007B0
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | 16:31:32.847758+03
    
```

Оказывается будущий мастер `replica1` и `replica2` подсоединены к прежнему мастеру. Время `reply_time` текущее. Ни один из кластеров мы пока не продвинули до мастера - все три кластера - физические реплики.

Поэтому в окне терминала периодически выводятся сообщения:

```

ОПЕРАТОР: SELECT slot_name, database, slot_type, xmin::text::int8, active,
pg_wal_lsn_diff(pg_current_wal_insert_lsn(), restart_lsn) AS retained_bytes FROM
pg_replication_slots LIMIT 50 OFFSET 0;
ОШИБКА: идёт процесс восстановления
ПОДСКАЗКА:  Функции управления WAL нельзя использовать в процессе восстановления.
    
```

14) Проверьте статистику слотов репликации на `replica1`, к которому подсоединен

БЫВШИЙ master:

```

postgres=# \c postgres postgres /var/run/postgresql 5433
Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в
"/var/run/postgresql", порт "5433".
postgres=# select * from pg_stat_replication \gx
-[ RECORD 1 ]-----+-----
pid           | 20280
usesysid      | 10
username      | postgres
application_name | master
client_addr    |
client_hostname |
client_port    | -1
backend_start  | 16:11:07.446672+03
backend_xmin   |
state         | streaming
sent_lsn      | 9/BC0007B0
write_lsn     | 9/BC0007B0
flush_lsn     | 9/BC0007B0
replay_lsn    | 9/BC0007B0
write_lag     |
flush_lag     |
replay_lag    |
sync_priority  | 0
sync_state    | async
reply_time    | 16:39:18.004533+03
    
```

Время `reply_time` текущее.

15) Посмотрите списки процессов экземпляров:

```

postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17/data/postmaster.pid`
  PID COMMAND
18615 postgres: master: checkpointer
18616 postgres: master: background writer
18617 postgres: master: startup recovering 0000000100000009000000BC
18624 postgres: master: walsender postgres [local] streaming 9/BC0007B0
18693 postgres: master: walsender postgres [local] streaming 9/BC0007B0
20279 postgres: master: walreceiver
postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17-replica/data1/postmaster.pid`
  PID COMMAND
18622 postgres: replica1: walreceiver
20280 postgres: replica1: walsender postgres [local] streaming 9/BC0007B0
25551 postgres: replica1: checkpointer
25552 postgres: replica1: background writer
25553 postgres: replica1: startup recovering 0000000100000009000000BC

postgres@tantor:~$ ps -o pid,command --ppid `head -n 1
/var/lib/postgresql/tantor-se-17-replica/data2/postmaster.pid`
  PID COMMAND
18692 postgres: replica2: walreceiver
24185 postgres: replica2: checkpointer
24186 postgres: replica2: background writer
24187 postgres: replica2: startup recovering 0000000100000009000000BC
    
```

Текущее состояние: replica1 **забирает** журналы с master. master **забирает** журналы с replica1. replica2 **забирает** журналы с master. Все кластеры **в режиме восстановления** (физические реплики).

Часть 7. Переключение на реплику

Как реплику сделать мастером? Можно продвинуть `replica1` до мастера командой:

а) `pg_ctl promote -D /var/lib/postgresql/tantor-se-17-replica/data1`

б) вызвав функцию `psql -p 5433 -c "select pg_promote();"`

Можно выбрать любой способ.

1) Продвиньте `replica1` до мастера:

```
postgres@tantor:~$ psql -p 5433 -c "select pg_promote();"
pg_promote
-----
 t
(1 row)
```

Сообщения в логах кластеров:

```
[25553] СООБЩЕНИЕ:   получен запрос повышения статуса
[18622] ВАЖНО: завершение процесса считывания журнала по команде администратора
[25553] СООБЩЕНИЕ:   записи REDO обработаны до смещения 9/BC000718, нагрузка системы: CPU: пользов.: 0.94 с,
система: 1.13 с, прошло: 104038.32 с
[25553] СООБЩЕНИЕ:   выбранный ID новой линии времени: 2
[25553] СООБЩЕНИЕ:   восстановление архива завершено
[25551] СООБЩЕНИЕ:   начата контрольная точка: force
[20279] СООБЩЕНИЕ:   репликация прекращена главным сервером
[20279] ПОДРОБНОСТИ: На линии времени 1 в 9/BC0007B0 достигнут конец журнала.
[20279] СООБЩЕНИЕ:   загрузка файла истории для линии времени 2 с главного сервера
[20279] ВАЖНО: завершение процесса считывания журнала по команде администратора
[18617] СООБЩЕНИЕ:   новая целевая линия времени 2
[25550] СООБЩЕНИЕ:   система БД готова принимать подключения
[25551] СООБЩЕНИЕ:   контрольная точка завершена: записано буферов: 2 (0.0%); добавлено файлов WAL 0, удалено: 0,
переработано: 0; запись=0.002 сек., синхр.=0.001 сек., всего=0.014 сек.; синхронизировано_файлов=2,
самая_долгая_синхр.=0.001 сек., средняя=0.001 сек.; расстояние=0 kB, ожидалось=0 kB; lsn=9/BC000828, lsn
redo=9/BC0007E0
[4727] СООБЩЕНИЕ:   начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени 2
[18617] СООБЩЕНИЕ:   запись REDO начинается со смещения 9/BC0007B0
[18692] СООБЩЕНИЕ:   репликация прекращена главным сервером
[18692] ПОДРОБНОСТИ: На линии времени 1 в 9/BC0007B0 достигнут конец журнала.
[18692] СООБЩЕНИЕ:   загрузка файла истории для линии времени 2 с главного сервера
[18692] ВАЖНО: завершение процесса считывания журнала по команде администратора
[24187] СООБЩЕНИЕ:   новая целевая линия времени 2
[4730] СООБЩЕНИЕ:   начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени 2
```

2) Посмотрите статус слотов репликации:

```
postgres@tantor:~$ psql -p 5433
```

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
master	t	9/BC000908	reserved
replica2	f		

(2 строки)

```
postgres=# select * from pg_stat_replication \gx
```

```
-[ RECORD 1 ]-----+-----
pid          | 4729
usesysid    | 10
username    | postgres
application_name | master
client_addr  |
client_hostname |
client_port  | -1
backend_start | 19:31:35.37509+03
backend_xmin  |
state       | streaming
sent_lsn    | 9/BC000908
write_lsn   | 9/BC000908
```

```
flush_lsn          | 9/BC000908
replay_lsn         | 9/BC000908
write_lag          |
flush_lag          |
replay_lag         |
sync_priority      | 0
sync_state         | async
reply_time         | 19:40:25.699932+03
```

```
postgres=# \c postgres postgres /var/run/postgresql 5432
```

Вы подключены к базе данных "postgres" как пользователь "postgres" через сокет в "/var/run/postgresql", порт "5432".

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
pgstandby1	f	0/19187E70	lost
replica1	f	9/BC0007B0	reserved
replica2	t	9/BC000908	reserved

(3 строки)

```
postgres=# select * from pg_stat_replication \gx
```

-[RECORD 1]-----	
pid	4731
usesysid	10
username	postgres
application_name	replica2
client_addr	
client_hostname	
client_port	-1
backend_start	19:31:35.411578+03
backend_xmin	
state	streaming
sent_lsn	9/BC000908
write_lsn	9/BC000908
flush_lsn	9/BC000908
replay_lsn	9/BC000908
write_lag	
flush_lag	
replay_lag	
sync_priority	0
sync_state	async
reply_time	19:45:05.821085+03

Новый мастер `replica1` передает журнальные данные физической реплике `master`.

Физическая реплика `master` передает журнальные данные физической реплике `replica1`.

В списке слотов бывшего мастера присутствует слот `replica1`, который был инициализирован, пока он был мастером. Названия слотов не зависят от названий кластеров. Кластеры неотличимы друг от друга и бывший мастер не может сопоставить то, что слот `replica1` использовался новым мастером. Это слот заставит `master` удерживать журнальные файлы для репликационного клиента, который вряд ли подсоединится, так как он теперь мастер.

Что хорошо: с использованием каскадирования можно сохранять журнальные файлы не на мастере, а на репликах, с которых другие реплики забирают журнальные записи.

3) При продвижении `replica1` до мастера, **линия времени увеличилась на единицу**. Это отражается в управляющих файлах и названиях файлов журналов. Также в директориях `PGDATA/pg_wal` кластеров были созданы текстовые файлы `00000002.history` в названии которых имеется **номер линии времени**.

Посмотрите содержимое файла истории:

```
postgres@tantor:~$ cat /var/lib/postgresql/tantor-se-17-
replica/data1/pg_wal/*.history
1          9/BC0007B0          no recovery target specified
```

4) Посмотрите линию времени в управляющем файле реплики `master` (у остальных кластеров то же самое):

```
postgres@tantor:~$ pg_controldata | grep timeline
postgres@tantor:~$ pg_controldata | grep времени
Линия времени последней конт. точки: 2
Пред. линия времени последней к. т.: 2
Линия времени мин. положения к. в.: 2
Формат хранения даты/времени: 64-битные целые
```

Этот файл будет использоваться в процессе восстановления с бэкапов, которые были созданы до появления новой линии времени.

5) **Неиспользуемые инициализированные слоты репликации всегда нужно удалять.**

Иначе эти слоты будут удерживать журналы до достижения `max_slot_wal_keep_size`, статус слота сменится на `unreserved`. Если после контрольной точки (файлы удаляются после прохождения контрольной точки) файлы журналов не удалятся, благодаря удержанию параметром `wal_keep_size`, то статус лота изменится на `extended`. Если удалятся, то статус слота изменится на `lost` и слот станет бесполезен.

Удалите слоты, которые не будете использовать:

```
postgres=# \c postgres postgres /var/run/postgresql 5432
Вы подключены к базе данных "postgres" как пользователь "postgres".
```

```
postgres=# select pg_drop_replication_slot('replica1');
pg_drop_replication_slot
```

```
-----
```

(1 строка)

```
postgres=# select pg_drop_replication_slot('pgstandby1');
pg_drop_replication_slot
```

```
-----
```

(1 строка)

6) Заранее создадим неинициализированный слот репликации для следующей смены ролей:

```
postgres=# select pg_create_physical_replication_slot('replica1');
pg_create_physical_replication_slot
```

```
-----
```

(replica1,)

(1 строка)

7) Проверьте список слотов:

```
postgres=# select slot_name, active, restart_lsn, wal_status from
pg_replication_slots;
```

slot_name	active	restart_lsn	wal_status
replica1	f		
replica2	t	9/BC000908	reserved

(2 строки)

Слот replica1 не инициализирован и не будет удерживать журналы.

Часть 8. Включение обратной связи

1) Если реплики планируется использовать для обслуживания запросов, то для защиты от сбоев долгих запросов на реплике можно настроить параметры, которые либо задержат применение журнальных записей на реплике, либо сообщат мастеру о том, что на реплике работают долгие запросы, и удалять старые версии строк не нужно.

Установите значение параметра `hot_standby_feedback=on` на реплике `master`:

```
postgres=# \c postgres postgres /var/run/postgresql 5432
Вы подключены к базе данных "postgres" как пользователь "postgres".
postgres=# alter system set hot_standby_feedback = on;
ALTER SYSTEM
postgres=# select pg_reload_conf();
 pg_reload_conf
-----
 t
(1 строка)
```

2) Протестировать обратную связь можно, открыв транзакцию на реплике `master`:

```
postgres=# begin transaction isolation level repeatable read;
BEGIN
postgres=*# select count(*) from pg_class;
 count
-----
 423
(1 строка)
```

Транзакция началась в момент начала выполнения `select`. Таблица, к которой обратился `select`, может быть любой.

3) В самих репликах искать процессы, выполняющие команды, удерживающие горизонт можно так же, как и на мастере - запросом к `pg_stat_activity`. Выполните на реплике запрос:

```
postgres=*# SELECT backend_xmin, backend_xid, pid, datname, state FROM
pg_stat_activity WHERE backend_xmin IS NOT NULL OR backend_xid IS NOT NULL ORDER
BY greatest(age(backend_xmin), age(backend_xid)) DESC;
 backend_xmin | backend_xid | pid | datname | state
-----+-----+-----+-----+-----
          17580 |              | 4117 | postgres | active
(1 строка)
```

4) В другом окне терминала, в другой сессии к кластеру `replica1`, являющемуся мастером:

```
postgres@tantor:~$ psql -p 5433
postgres=# select slot_name, active, active_pid, xmin from pg_replication_slots;
 slot_name | active | active_pid | xmin
-----+-----+-----+-----
 master   | t      |          4729 | 17580
 replica2 | f      |              |
(2 строки)
```

Реплика удерживает горизонт всех баз данных кластера (на мастере не могут удаляться вакуумом старые версии строк) на `xid=17580`.

5) Получите номер транзакции:

```
postgres=# select pg_current_xact_id();
 pg_current_xact_id
```

17580

(1 строка)

6) Можно выполнить транзакции, но достаточно просто увеличить счетчик транзакций.

Получите номер транзакции, и счетчик транзакций увеличится:

```
postgres=# select pg_current_xact_id();
pg_current_xact_id
```

17581

(1 строка)

7) Выполните на мастере запрос, который покажет, какой серверный процесс экземпляра мастера удерживает горизонт:

```
postgres=# SELECT backend_xmin, backend_xid, pid, datname, state FROM
pg_stat_activity WHERE backend_xmin IS NOT NULL OR backend_xid IS NOT NULL ORDER
BY greatest(age(backend_xmin), age(backend_xid)) DESC;
```

backend_xmin	backend_xid	pid	datname	state
17582		6562	postgres	active

(1 row)

`pg_stat_activity` показывает только процессы своего экземпляра.

8) Выполните команду:

```
postgres=# select slot_name, active, active_pid, xmin from pg_replication_slots;
```

slot_name	active	active_pid	xmin
master	t	4729	17580
replica2	f		

(2 строки)

Горизонт всех баз данных мастера (текущий мастер - это replica1) удерживается на `xid=17580`.

9) Подсоединитесь к реплике и посмотрите данные о процессах реплики:

```
postgres=# \c postgres postgres /var/run/postgresql 5432
```

You are now connected to database "postgres" as user "postgres" via socket in "/var/run/postgresql" at port "5432".

```
postgres=# SELECT backend_xmin, backend_xid, pid, datname, state FROM
pg_stat_activity WHERE backend_xmin IS NOT NULL OR backend_xid IS NOT NULL ORDER
BY greatest(age(backend_xmin), age(backend_xid)) DESC;
```

backend_xmin	backend_xid	pid	datname	state
17580		4117	postgres	idle in transaction
17582		7692	postgres	active

(2 rows)

4117 - pid серверного процесса, в котором открыта транзакция.

7692 - pid серверного процесса на master, в котором выполнялся этот запрос. 17582

означает, что этот серверный процесс выдаёт актуальные данные (в соответствии с изменениями, приехавшими с мастера и примененными к реплике).

7) Завершите открытую транзакцию в том окне, где она открыта:

```
postgres=# commit;
COMMIT
```

8) В пределах 10 секунд (значение параметра `walreceiver_status_interval`) `xmin` на `replica1` перестанет удерживаться и `xmin` увеличится:

```
postgres=# select slot_name, active, active_pid, xmin from pg_replication_slots;
 slot_name | active | active_pid | xmin
-----+-----+-----+-----
 master   | t      |          4729 | 17582
 replica2 | f      |              |
(2 строки)
```

4729 - pid walsender на replica1. Это можно проверить командой:

```
postgres@tantor:~$ ps -ef | grep 4729
postgres 4729 25550 postgres: replica1: walsender postgres [local] streaming
9/BC0017B8
```

Часть 9. Утилита pg_rewind

1) Остановите replical:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17-replica/data1
ожидание завершения работы сервера...
[25550] СООБЩЕНИЕ: получен запрос на быстрое выключение
[25550] СООБЩЕНИЕ: прерывание всех активных транзакций
[25550] СООБЩЕНИЕ: фоновый процесс "logical replication launcher" (PID 4728) завершился с кодом
выхода 1
[25551] СООБЩЕНИЕ: выключение
[25551] СООБЩЕНИЕ: начата контрольная точка: shutdown immediate
[25551] СООБЩЕНИЕ: контрольная точка завершена: записано буферов: 0 (0.0%); добавлено файлов WAL
0, удалено: 0, переработано: 0; запись=0.001 сек., синхр.=0.001 сек., всего=0.012 сек.;
синхронизировано_файлов=0, самая_долгая_синхр.=0.000 сек., средняя=0.000 сек.; расстояние=0 кВ,
ожидалось=0 кВ; lsn=9/BC001950, lsn redo=9/BC001950
[25550] СООБЩЕНИЕ: система БД выключена
готово
сервер остановлен
```

2) Подсоединитесь к master (порт 5432) и повысьте его до мастера:

```
postgres@tantor:~$ psql
postgres=# select pg_promote();

LOG: replication terminated by primary server
DETAIL: End of WAL reached on timeline 2 at 9/BC001A18.
LOG: fetching timeline history file for timeline 3 from primary server
FATAL: terminating walreceiver process due to administrator command
LOG: new target timeline is 3
СООБЩЕНИЕ: неверная длина записи в позиции 9/BC001B40: ожидалось минимум 26, получено 0
СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии
времени 3

 pg_promote
-----
 t
(1 строка)
```

3) Сейчас master является мастером, к нему подсоединена replica2, которая не останавливалась и принимает журнальные данные.

Мы корректно остановили replical до продвижения нового мастера. Можем ли мы запустить replical или нужно ещё что-то сделать?

В демонстрационных целях запустим и остановим replical. Это эквивалентно тому, как если бы мы забыли остановить replical до продвижения master, или тому, как если бы экземпляр replical некорректно остановился и не успел бы передать последнюю журнальную запись на master.

```
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1
ожидание запуска сервера....
СООБЩЕНИЕ: запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0,
64-bit
СООБЩЕНИЕ: для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5433
СООБЩЕНИЕ: для приёма подключений по адресу IPv6 ":::" открыт порт 5433
СООБЩЕНИЕ: для приёма подключений открыт Unix-сокеты "/var/run/postgresql/.s.PGSQL.5433"
СООБЩЕНИЕ: система БД была выключена:
СООБЩЕНИЕ: система БД готова принимать подключения
готово
сервер запущен
```

4) Остановите replical:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17-replica/data1
```

```

ожидание завершения работы сервера....
СООБЩЕНИЕ: получен запрос на быстрое выключение
СООБЩЕНИЕ: прерывание всех активных транзакций
СООБЩЕНИЕ: фоновый процесс "logical replication launcher" (PID 27234) завершился с кодом выхода 1
СООБЩЕНИЕ: выключение
СООБЩЕНИЕ: начата контрольная точка: shutdown immediate
СООБЩЕНИЕ: контрольная точка завершена: записано буферов: 3 (0.0%); добавлено файлов WAL 0, удалено: 0,
переработано: 0; запись=0.001 сек., синхр.=0.001 сек., всего=0.004 сек.; синхронизировано файлов=2,
самая долгая_синхр.=0.001 сек., средняя=0.001 сек.; расстояние=0 kB, ожидалось=0 kB; lsn=9/BC001A30, lsn
redo=9/BC001A30
СООБЩЕНИЕ: система БД выключена
    готово
сервер остановлен
    
```

5) Перед запуском мы не создали файл `standby.signal` и кластер запустился с ролью мастера.

Создайте файл `standby.signal`:

```

postgres@tantor:~$
touch /var/lib/postgresql/tantor-se-17-replica/data1/standby.signal
    
```

Но уже поздно: при остановке **была выполнена контрольная точка** и создана журнальная запись на линии времени 2.

Если сейчас снова запустить экземпляр `replica1`, то `master` будет отказывать ему в доступе, `replica1` будет переподсоединяться к `master` без задержки, и **безостановочно** писать сообщения в диагностический лог. Пример таких сообщений:

```

СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии
времени 2
СООБЩЕНИЕ: репликация прекращена главным сервером
ПОДРОБНОСТИ: На линии времени 2 в 9/BC0019E8 достигнут конец журнала.
ВАЖНО: завершение процесса считывания журнала по команде администратора
СООБЩЕНИЕ: новая линия времени 3 ответвилась от текущей линии времени базы данных 2 до
текущей точки восстановления 9/BC001AC8
СООБЩЕНИЕ: waiting for WAL to become available at 9/BC001AE2
    
```

В таком случае можно попробовать использовать утилиту `pg_rewind`.

6) Дайте команду:

```

postgres@tantor:~$ pg_rewind -D /var/lib/postgresql/tantor-se-17-replica/data1 --
source-server='user=postgres port=5432' -R -P
pg_rewind: подключение к серверу установлено
pg_rewind: ошибка: на целевом сервере должны быть контрольные суммы данных или
"wal_log_hints = on"
    
```

Если утилита не выдала ошибку о том, что на `replica1` не включен подсчёт контрольных сумм (как и на остальных кластерах), и успешно выполнялась, то можно перейти к запуску экземпляра `replica1`.

7) Убедитесь, что подсчета контрольных сумм нет:

```

postgres@tantor:~$ pg_checksums -D /var/lib/postgresql/tantor-se-17-replica/data1
pg_checksums: ошибка: контрольные суммы в кластере не включены
    
```

8) Включите подсчёт контрольных сумм:

```

postgres@tantor:~$ pg_checksums -e -D /var/lib/postgresql/tantor-se-17-
replica/data1
    
```

Если утилита выдала ошибку типа:

```
pg_checksums: ошибка: неверный номер сегмента 0 в имени файла
"/var/lib/postgresql/tantor-se-17-replica/data1/global/pg_store_plans.stat"
```

Это означает, то в файле табличного пространства присутствует файл, который не должен там находиться. Ошибки связанные с присутствием неизвестных файлов в PGDATA возможны. В этом примере это файл неизвестного формата:

```
pg_store_plans.c
/* Location of stats file */
#define PGSP_DUMP_FILE    "global/pg_store_plans.stat"
```

9) В любом случае утилита `pg_rewind` скопирует нужные файлы с мастера, поэтому удалите файл, из-за которого не включается подсчет контрольных сумм на блоках файлов данных:

```
postgres@tantor:~$ rm /var/lib/postgresql/tantor-se-17-
replica/data1/global/pg_store_plans.stat
```

10) Повторите команду включения подсчета контрольных сумм:

```
postgres@tantor:~$ pg_checksums -e -D /var/lib/postgresql/tantor-se-17-
replica/data1
```

```
Обработка контрольных сумм завершена
Просканировано файлов: 1913
Просканировано блоков: 54747
Записано файлов: 1563
Записано блоков: 54701
pg_checksums: синхронизация каталога данных
pg_checksums: модификация управляющего файла
Контрольные суммы в кластере включены
```

11) Запустите `pg_rewind` ещё раз:

```
postgres@tantor:~$ pg_rewind -D /var/lib/postgresql/tantor-se-17-replica/data1 --
source-server='user=postgres port=5432' -R -P
```

```
pg_rewind: подключение к серверу установлено
pg_rewind: серверы разошлись в позиции WAL 9/BC0019E8 на линии времени 2
pg_rewind: перемотка от последней общей контрольной точки в позиции 9/BC001950 на
линии времени 2
pg_rewind: чтение списка исходных файлов
pg_rewind: чтение списка целевых файлов
pg_rewind: чтение WAL в целевом кластере
pg_rewind: требуется скопировать 194 МБ (общий размер исходного каталога: 615 МБ)
199097/199097 КБ (100%) скопировано
pg_rewind: создание метки копии и модификация управляющего файла
pg_rewind: синхронизация целевого каталога данных
pg_rewind: Готово!
```

Можно ли запускать экземпляр кластера? **Нельзя**. Утилита `pg_rewind` синхронизировала все файлы с мастером, в том числе файлы параметров конфигурации, и они содержат настройки мастера.

До **запуска** утилиты `pg_rewind` **стоит** **сохранять** **файлы** **параметров**, **которые** **лежат** **в** **PGDATA**.

12) Отредактируйте файл `postgresql.auto.conf`, приведя его к такому виду:

```
postgres@tantor:~$ cat /var/lib/postgresql/tantor-se-17-
replica/data1/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
listen_addresses = '*'
max_slot_wal_keep_size = '128MB'
max_wal_size = '128MB'
min_wal_size = '512MB'
idle_in_transaction_session_timeout = '100min'
cluster_name = 'replica1'
primary_slot_name = 'replica1'
logging_collector = 'off'
hot_standby_feedback = 'on'
primary_conninfo = 'user=postgres port=5432'
wal_retrieve_retry_interval = '30s'
port = 5433
```

13) Запустите экземпляр replica1:

```
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1
```

ожидание запуска сервера....

```
[18861] СООБЩЕНИЕ: запускается PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3)
12.2.0, 64-bit
[18861] СООБЩЕНИЕ: для приёма подключений по адресу IPv4 "0.0.0.0" открыт порт 5433
[18861] СООБЩЕНИЕ: для приёма подключений по адресу IPv6 ":::" открыт порт 5433
[18861] СООБЩЕНИЕ: для приёма подключений открыт Unix-сокет "/var/run/postgresql/.s.PGSQL.5433"
[18864] СООБЩЕНИЕ: работа системы БД была прервана в процессе восстановления, время в журнале:
[18864] ПОДСКАЗКА: Если это происходит постоянно, возможно, какие-то данные были испорчены и для восстановления
стоит выбрать более раннюю точку.
[18864] СООБЩЕНИЕ: переход в режим резервного сервера
[18864] СООБЩЕНИЕ: запись REDO начинается со смещения 9/BC0019E8
[18864] СООБЩЕНИЕ: согласованное состояние восстановления достигнуто в позиции 9/BC001D48
[18864] СООБЩЕНИЕ: неверная длина записи в позиции 9/BC001D48: ожидалось минимум 26, получено 0
[18861] СООБЩЕНИЕ: система БД готова принимать подключения в режиме "только чтение"
[18865] СООБЩЕНИЕ: начало передачи журнала с главного сервера, с позиции 9/BC000000 на линии времени 3
готово
сервер запущен
```

14) Проверьте статистику репликации на мастере:

```
postgres@tantor:~$ psql
postgres=# select * from pg_stat_replication \gx
-[ RECORD 1 ]-----+-----
pid          | 23531
usesysid     | 10
username     | postgres
application_name | replica2
client_addr  |
client_hostname |
client_port  | -1
backend_start | 12:32:18.89956+03
backend_xmin  |
state        | streaming
sent_lsn     | 9/BC001D48
write_lsn    | 9/BC001D48
flush_lsn    | 9/BC001D48
replay_lsn   | 9/BC001D48
write_lag    |
flush_lag    |
replay_lag   |
sync_priority | 0
sync_state   | async
reply_time   | 13:17:04.270387+03
-[ RECORD 2 ]-----+-----
pid          | 18866
usesysid     | 10
username     | postgres
```

```

application_name | replica1
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 13:13:42.353704+03
backend_xmin     |
state            | streaming
sent_lsn         | 9/BC001D48
write_lsn        | 9/BC001D48
flush_lsn        | 9/BC001D48
replay_lsn       | 9/BC001D48
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | 13:17:02.430073+03
    
```

Обе реплики принимают журнальные данные и накатывают их.

Напоминание: контрольные суммы включены только на `replica1`. Обратная связь (`hot_standby_feedback = 'on'`) включена на `master` и `replica1`.

Глава 8b. Логическая репликация

Часть 1. Репликация таблицы

1) Повысьте реплику на порту 5433 до мастера:

```
postgres@tantor:~$ psql -p 5433 -c "select pg_promote()"
pg_promote
-----
t
(1 строка)
```

Сейчас два мастера на портах 5432 и 5433. У мастера на порту 5432 есть реплика на порту 5434.

Пример сообщений в логе о повышении реплики:

```
[18864] СООБЩЕНИЕ: получен запрос повышения статуса
[9445] ВАЖНО: завершение процесса считывания журнала по команде администратора
[18864] СООБЩЕНИЕ: неверная длина записи в позиции 9/BC09D2D0: ожидалось минимум 26, получено 0
[18864] СООБЩЕНИЕ: записи REDO обработаны до смещения 9/BC09D288, нагрузка системы: CPU: пользов.: 0.09 с,
система: 0.11 с, прошло: 17564.43 с
[18864] СООБЩЕНИЕ: последняя завершённая транзакция была выполнена в 18:03:37.805708+03
[18864
```

5) Проверьте, что строки добавились:

```
] СООБЩЕНИЕ: выбранный ID новой линии времени: 4
[18864] СООБЩЕНИЕ: восстановление архива завершено
[18862] СООБЩЕНИЕ: начата контрольная точка: force
[18861] СООБЩЕНИЕ: система БД готова принимать подключения
[18862] СООБЩЕНИЕ: контрольная точка завершена: записано буферов: 7 (0.0%); добавлено файлов WAL 0, удалено: 0,
переработано: 0; запись=0.504 сек., синхр.=0.004 сек., всего=0.510 сек.; синхронизировано_файлов=7,
самая_долгая_синхр.=0.002 сек., средняя=0.001 сек.; расстояние=29 kB, ожидалось=159 kB; lsn=9/BC09D3C8, lsn
redo=9/BC09D338
```

2) Удалите слот репликации `replica1` в кластере `master` (порт 5432), который использовался `replica1` (порт 5433) и уже не нужен:

```
psql -p 5432 -c "select slot_name, slot_type, active, restart_lsn, wal_status
from pg_replication_slots"
```

```
 slot_name | slot_type | active | restart_lsn | wal_status
-----+-----+-----+-----+-----
 replica1 | physical | f      | 9/BC09D2D0  | reserved
 replica2 | physical | t      | 9/BC09D3F8  | reserved
(2 строки)
```

```
psql -p 5432 -c "select pg_drop_replication_slot('replica1')"
```

```
pg_drop_replication_slot
-----
```

(1 строка)

```
psql -p 5432 -c "select slot_name, slot_type, active, restart_lsn, wal_status
from pg_replication_slots"
```

```
 slot_name | slot_type | active | restart_lsn | wal_status
-----+-----+-----+-----+-----
 replica2 | physical | t      | 9/BC09D3F8  | reserved
(1 строка)
```

На `replica2` есть два неинициализированных слота репликации:

```
psql -p 5434 -c "select slot_name, slot_type, active, restart_lsn, wal_status
from pg_replication_slots"
```

```
 slot_name | slot_type | active | restart_lsn | wal_status
-----+-----+-----+-----+-----
 master   | physical | f      |              |
 replica1 | physical | f      |              |
(2 строки)
```

3) Проверьте какие таблицы в базе postgres на порту 5432 не имеют репликационного идентификатора:

```
psql -p 5432 -c "SELECT relnamespace::regnamespace||'.'||relname "table"
FROM pg_class
WHERE relreplident IN ('d','n') -- d первичный ключ, n никакие
AND relkind IN ('r','p') -- r таблица, p секционированная
AND oid NOT IN (SELECT indrelid FROM pg_index WHERE indisprimary)
AND relnamespace <> 'pg_catalog'::regnamespace
AND relnamespace <> 'information_schema'::regnamespace
ORDER BY 1"
table
-----
public.demo2
public.hypo
utl_file.utl_file_dir
(3 строки)
```

4) Удалите таблицы demo2, t если они существуют:

```
psql -p 5432 -c "drop table if exists t"
NOTICE: table "t" does not exist, skipping
DROP TABLE
psql -p 5432 -c "drop table if exists demo2"
DROP TABLE
```

5) Создайте таблицу, которую будем реплицировать, и вставьте строку:

```
psql -p 5432 -c "create table t (id bigserial PRIMARY KEY, t text)"
CREATE TABLE

psql -p 5432 -c "insert into t (t) values ('a')"
INSERT 0 1
```

6) Создайте определение таблицы-приемника в базе postgres кластера на порту 5433:

```
pg_dump -t t --schema-only --clean --if-exists | psql -p 5433
```

Этот шаг обязательный: структуру таблицы, в которую будут реплицироваться изменения, нужно создавать отдельно, так как в функционале логической репликации нет автоматического создания таблиц. Таблица и столбцы должны иметь те же названия. Порядок столбцов не важен, могут быть дополнительные столбцы, наличие которых не мешало бы вставлять строки.

Препятствие для вставки строк: наличие ограничения целостности NOT NULL в отсутствие значения по умолчанию DEFAULT.

7) Установите wal_level=logical на всех кластерах.

Установите hot_standby_feedback=on на replica2, он не был установлен в предыдущей практике. **Включение обратной связи обязательно для логической репликации**, в противном случае при изменении определения и набора таблиц в публикации (то, что меняет строки в

таблицах системного каталога, которые хранят данные о свойствах реплицируемых таблиц)

могут возникать ошибки "Этот слот был аннулирован из-за конфликта с восстановлением".

Установите `checkpoint_timeout='30min'`, чтобы контрольные точки и точки рестарта (по умолчанию, раз в 5 минут) не писали сообщения в логи кластеров, затрудняя чтение сообщений логической репликации:

```
psql -p 5432 -c "ALTER SYSTEM SET wal_level=logical"
psql -p 5433 -c "ALTER SYSTEM SET wal_level=logical"
psql -p 5434 -c "ALTER SYSTEM SET wal_level=logical"
psql -p 5434 -c "ALTER SYSTEM SET hot_standby_feedback=on"
psql -p 5432 -c "alter system set checkpoint_timeout='30min'"
psql -p 5433 -c "alter system set checkpoint_timeout='30min'"
psql -p 5434 -c "alter system set checkpoint_timeout='30min'"
```

Изменение параметра `wal_level` требует рестарта экземпляров. **Убедитесь** в этом:

```
psql -p 5432 -c "select pg_reload_conf()"
pg_reload_conf
-----
t
(1 строка)
```

```
psql -p 5432 -c "select * from pg_settings where name = 'wal_level'" -x
-[ RECORD 1 ]-----+-----
name           | wal_level
setting        | replica
unit           |
category       | Write-Ahead Log / Settings
short_desc     | Sets the level of information written to the WAL.
extra_desc     |
context        | postmaster
vartype        | enum
source         | default
min_val        |
max_val        |
enumvals       | {minimal,replica,logical}
boot_val       | replica
reset_val      | replica
sourcefile     |
sourceline     |
pending_restart | t
```

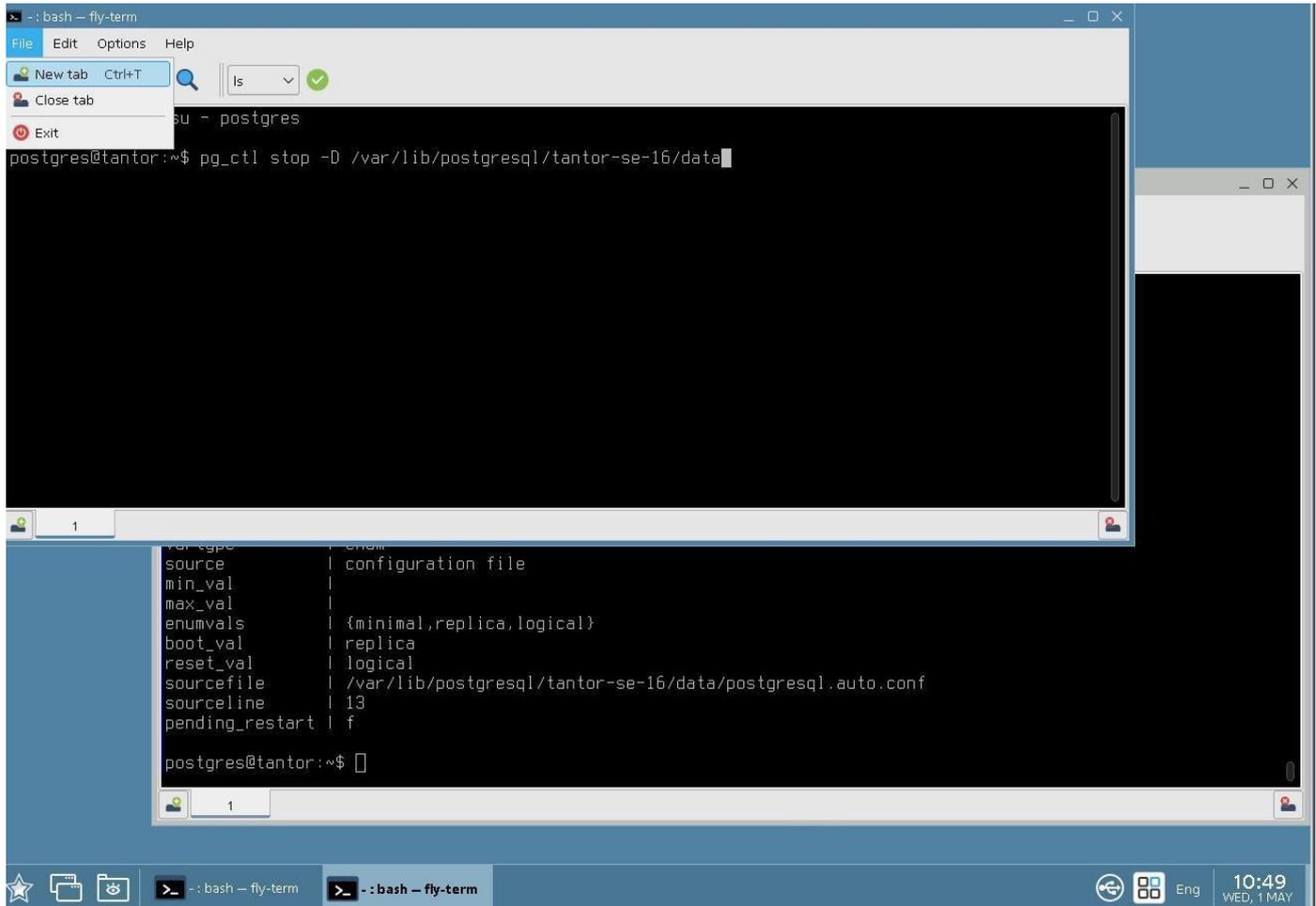
8) Остановите и запустите экземпляры в окне (окнах) терминала, в котором (которых) хотите получать диагностические сообщения:

```
pg_ctl stop -D /var/lib/postgresql/tantor-se-17-replica/data2
pg_ctl stop -D /var/lib/postgresql/tantor-se-17-replica/data1
pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data

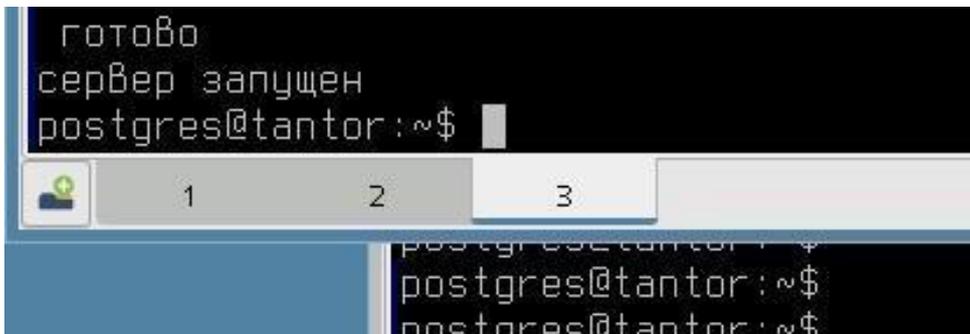
pg_ctl start -D /var/lib/postgresql/tantor-se-17/data
pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1
pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data2
```

Можно использовать несколько окон терминала, чтобы было удобнее видеть, от какого экземпляра какие сообщения выводятся. В одном окне терминала сложно отличить сообщения от разных экземпляров. Три терминала удобно открыть как закладки, а не окна. Для открытия

терминала в виде закладки можно выбрать в меню терминала File -> New, или комбинацию клавиш <Ctrl+t>:



Ссылки для переключения (1 2 3) находятся слева внизу окна терминала:



9) Создайте публикацию для таблицы t:

```
psql -p 5432 -c "CREATE PUBLICATION t for TABLE t"
CREATE PUBLICATION
```

10) Создайте подписку на `replica1`, подсоединяющуюся к **физической реплике** `replica2`. Подсоединение к физической реплике усложняет топологию, но уменьшает нагрузку на мастер. Имя подписки определяет по умолчанию имя логического слота репликации и должно быть уникальным во всей конфигурации:

```
psql -p 5433 -c "CREATE SUBSCRIPTION sub1 CONNECTION 'dbname=postgres port=5434
user=postgres' PUBLICATION t WITH (origin=none)"
```

Если бы подписка подсоединялась напрямую к мастеру, то команда выдала бы результат сразу. У нас подписка подсоединена к физической реплике и команда создания подписки подвиснет. Через 15-17 секунд команда отвиснет и выдаст следующий результат:

ЗАМЕЧАНИЕ: на сервере публикации создан слот репликации "sub1"
 CREATE SUBSCRIPTION

Если команда создания подписки висит больше 20 секунд, то это означает, что никакой активности фоновых и серверных процессов на мастере нет, в этом случае выполните следующий пункт практики. При реальной работе транзакции на мастере присутствуют команды создания подписок и команды начальной синхронизации будут выполняться с задержкой до ~20 секунд.

11) Если команда создания подписки висит и не выдаёт результат, то в любом другом окне терминала можно дать команду:

```
psql -p 5432 -c "select pg_log_standby_snapshot()"
pg_log_standby_snapshot
-----
9/BC0D9C58
(1 строка)
```

В логе кластеров видны **команды репликационного протокола**:

```
[12447] СООБЩЕНИЕ: процесс логического декодирования достиг точки
согласованности в 9/BC0D9C10
[12447] ПОДРОБНОСТИ: Больше активных транзакций нет.
ОПЕРАТОР: CREATE_REPLICATION_SLOT "sub1" LOGICAL pgoutput (SNAPSHOT 'nothing')
[12551] СООБЩЕНИЕ: запускается применяющий процесс логической репликации для
подписки "sub1"
[12552] СООБЩЕНИЕ: начинается логическое декодирование для слота "sub1"
[12552] ПОДРОБНОСТИ: Передача транзакций, фиксируемых после 9/BC0D9C58, чтение
WAL с 9/BC0D9C10.
[12552] ОПЕРАТОР: START_REPLICATION_SLOT "sub1" LOGICAL 0/0 (proto_version '4',
origin 'none', publication_names 't')
```

После следующего сообщения команда создания подписки выдаст результат:

```
[12552] СООБЩЕНИЕ: процесс логического декодирования достиг точки
согласованности в 9/BC0D9C10
[12552] ПОДРОБНОСТИ: Больше активных транзакций нет.
[12552] ОПЕРАТОР: START_REPLICATION_SLOT "sub1" LOGICAL 0/0 (proto_version '4',
origin 'none', publication_names 't')
[12553] СООБЩЕНИЕ: процесс синхронизации таблицы при логической репликации для
подписки "sub1", таблицы "t" запущен
```

Процессы 12551 и 12553 - это процессы replical.

Остальные процессы - это процессы replica2.

12) **Процесс синхронизации запустился**, но в отсутствие транзакций на мастере, будет висеть и ожидать точки синхронизации. Для ускорения синхронизации вызовите функцию `pg_log_standby_snapshot()` на мастере ещё раз, либо перейдите к 13-му пункту, в котором выполняется транзакция на мастере:

```
psql -p 5433 -c "select * from t"
 id | t
----+---
(0 строк)
```

```
psql -p 5432 -c "select pg_log_standby_snapshot()"
pg_log_standby_snapshot
-----
9/BC1CADE0
(1 строка)
```

После вызова функции (либо контрольной точки, либо через некоторое время после появления любой транзакции на мастере и передачи журнальной записи на физическую реплику) появится сообщение:

```
[12601] СООБЩЕНИЕ: процесс синхронизации таблицы при логической репликации для подписки "sub1", таблицы "t" закончил обработку
```

В логе физической реплики появятся сообщения о том, что слот достиг точки согласованности и готов к работе:

```
СООБЩЕНИЕ: процесс логического декодирования достиг точки согласованности в 9/BC263F10
ПОДРОБНОСТИ: Больше активных транзакций нет.
ОПЕРАТОР: CREATE_REPLICATION_SLOT "pg_43351_sync_43342_7353194261070147214" LOGICAL pgoutput (SNAPSHOT 'use')
СООБЩЕНИЕ: начинается логическое декодирование для слота "pg_43351_sync_43342_7353194261070147214"
ПОДРОБНОСТИ: Передача транзакций, фиксируемых после 9/BC263F58, чтение WAL с 9/BC263E48.
ОПЕРАТОР: START_REPLICATION SLOT "pg_43351_sync_43342_7353194261070147214" LOGICAL 9/BC263F58 (proto_version '4', origin 'none', publication_names '"t"')
```

```
psql -p 5433 -c "select * from t"
 id | t
----+---
  1 | a
(1 строка)
```

13) Проверьте, что репликация идёт:

```
psql -p 5432 -c "INSERT INTO t (t) VALUES ('b')"
```

INSERT 0 1

```
psql -p 5433 -c "select * from t"
 id | t
----+---
  1 | a
  2 | b
(2 rows)
```

Вставили в мастер вторую строку, подписка подсоединенная получила эту строку с физической реплики.

Примечание к этой части практики: далее приводится список команд, которые позволяют повторить создание подписки (начиная с пункта 4 до 12-го пункта):

```
psql -p 5432 -c "checkpoint"
psql -p 5433 -c "drop SUBSCRIPTION sub1"
psql -p 5432 -c "drop PUBLICATION t"
psql -p 5432 -c "select pg_log_standby_snapshot()"
psql -p 5432 -c "drop table t"
psql -p 5432 -c "create table t (id bigserial PRIMARY KEY, t text)"
```

```
psql -p 5432 -c "insert into t (t) values ('a')"  
pg_dump -t t --schema-only --clean --if-exists | psql -p 5433 > /dev/null  
psql -p 5432 -c "CREATE PUBLICATION t for TABLE t"  
psql -p 5433 -c "CREATE SUBSCRIPTION sub1 CONNECTION 'dbname=postgres port=5434  
user=postgres' PUBLICATION t WITH (origin=none)"  
  
psql -p 5433 -c "select * from t"  
psql -p 5432 -c "INSERT INTO t (t) VALUES ('b')"  
psql -p 5433 -c "select * from t"
```

Часть 2. Репликация без первичного ключа

1) Удалите первичный ключ таблицы `t` на источнике (порт 5432):

```
psql -c "ALTER TABLE t DROP CONSTRAINT t_pkey"
ALTER TABLE
```

Можно было бы удалить ограничение целостности и на таблице-приёмнике, но не будем этого делать, так как позже снова добавим первичный ключ.

Если на источнике не вносить дублирующиеся строки, то на приёмнике ограничение целостности себя не будет проявлять. Если внести дубликат в столбце `id`, то применение записей в подписке приостановится.

```
psql -c "\d t"
```

Таблица "public.t"				
Столбец	Тип	Правило сортировки	Допустимость NULL	По умолчанию
id	bigint		not null	nextval('t_id_seq'::regclass)
t	text			

```
Публикации:
"t"
```

```
psql -p 5433 -c "\d t"
```

Таблица "public.t"				
Столбец	Тип	Правило сортировки	Допустимость NULL	По умолчанию
id	bigint		not null	nextval('t_id_seq'::regclass)
t	text			

```
Индексы:
"t_pkey" PRIMARY KEY, btree (id)
```

Последовательности для генерации значения столбца `id` и ограничение целостности **NOT NULL** сохранились и присутствуют в обеих таблицах.

2) Ключа на таблице нет. Проверим, что вставки строк не блокируются и корректно реплицируются. Вставьте строку в таблицу `t`:

```
psql -c "INSERT INTO t (t) VALUES ('b')"
INSERT 0 1
```

3) Обновления и удаления блокируются. Дайте команды обновления и удаления строки и посмотрите какая ошибка выдаётся:

```
psql -c "update t set t='c' where id=2"
```

ОШИБКА: изменение в таблице "t" невозможно, так как в ней **отсутствует идентификатор реплики**, но она публикует изменения

ПОДСКАЗКА: Чтобы эта таблица поддерживала изменение, **установите REPLICIA IDENTITY, выполнив ALTER TABLE.**

```
psql -c "delete from t where id=2"
```

ОШИБКА: удаление из таблицы "t" невозможно, так как в ней **отсутствует идентификатор реплики**, но она публикует удаления

ПОДСКАЗКА: Чтобы эта таблица поддерживала удаление, **установите REPLICIA IDENTITY, выполнив ALTER TABLE.**

4) Установите идентификаторами строк **все столбцы**:

```
psql -c "ALTER TABLE t REPLICA IDENTITY FULL"
ALTER TABLE
```

5) Обновления и удаления теперь не блокируются и корректно реплицируются. Выполните команды:

```
psql -c "update t set t='c' where id=3"
UPDATE 1
psql -c "delete from t where id=3"
DELETE 1
psql -p 5432 -c "select * from t"
 id | t
----+----
  1 | a
  2 | b
(2 строки)
psql -p 5433 -c "select * from t"
 id | t
----+----
  1 | a
  2 | b
(2 строки)
```

Команды вставки и удаления реплицировались корректно.

Использование **REPLICA IDENTITY FULL** нежелательно тем, что при выполнении на источнике UPDATE и DELETE, через журнал передаются значения всех столбцов, а это увеличивает трафик. Если фактически можно идентифицировать строки по нескольким столбцам, то стоит их использовать в качестве идентификатора - первичного ключа.

6) Посмотрим что будет, если установить способ идентификации строк в **NOTHING**.

Выполните команду:

```
psql -c "ALTER TABLE t REPLICA IDENTITY NOTHING"
ALTER TABLE
```

7) Выполните команду обновления строки:

```
psql -c "update t set t='c' where id =2"
```

ОШИБКА: изменение в таблице "t" невозможно, так как в ней **отсутствует идентификатор реплики**, но она публикует изменения

ПОДСКАЗКА: Чтобы эта таблица поддерживала изменение, **установите REPLICA IDENTITY, выполнив ALTER TABLE**.

Ошибка такая же, как была прежде: нет идентификатора для опубликования обновления или удаления.

8) Ни добавление первичного ключа, ни REFRESH, ни **DISABLE** подписки не устранят ошибку. В следующих пунктах проверим это.

Приостановите подписку:

```
psql -p 5433 -c "ALTER SUBSCRIPTION sub1 DISABLE"
ALTER SUBSCRIPTION
```

9) Посмотрите статус подписки командой psql \dRs:

```
psql -p 5433 -c "\dRs"
Список подписок
```

```
Имя | Владелец | Включён | Публикация
-----+-----+-----+-----
sub1 | postgres | f       | {t}
(1 строка)
```

10) На источнике попробуйте обновить строку:

```
psql -c "update t set t='c' where id =2"
```

ОШИБКА: изменение в таблице "t" невозможно, так как в ней **отсутствует идентификатор реплики**, но она публикует изменения

ПОДСКАЗКА: Чтобы эта таблица поддерживала изменение, **установите REPLICATION IDENTITY**, выполнив **ALTER TABLE**.

Обновление не проходит, хоть подписка и приостановлена.

11) Вставьте строку на источнике:

```
psql -c "INSERT INTO t (t) VALUES ('c')"
```

INSERT 0 1

12) Проверьте, что на приемнике строка не появилась:

```
psql -p 5433 -c "select * from t"
```

id	t
1	a
2	b

(2 строки)

13) **Включите** подписку:

```
psql -p 5433 -c "ALTER SUBSCRIPTION sub1 ENABLE"
```

ALTER SUBSCRIPTION

14) Проверьте, что на приемнике строка **появилась**:

```
psql -p 5433 -c "select * from t"
```

id	t
1	a
2	b
4	c

(3 строки)

После приостановки подписки (перевод в статус `DISABLE`) применение изменений приостановилось. После включения (перевод в статус `ENABLE`) - накопленные изменения были применены.

15) Добавьте первичный ключ:

```
psql -c "ALTER TABLE t ADD CONSTRAINT t_key PRIMARY KEY (id)"
```

ALTER TABLE

Но его наличия не достаточно. Нужно указать, чтобы он использовался как `REPLICATION IDENTITY`, так как до этого мы устанавливали `REPLICATION IDENTITY NOTHING`.

16) Включите использование первичного ключа в качестве идентификатора репликации, то есть установите значение **по умолчанию**:

```
psql -c "ALTER TABLE t REPLICA IDENTITY DEFAULT"  
ALTER TABLE
```

17) Теперь обновление строки не выдаёт ошибки и репликация проходит:

```
psql -c "update t set t='d' where id =4"  
UPDATE 1
```

```
psql -p 5433 -c "select * from t"
```

```
 id | t  
----+---  
  1 | a  
  2 | b  
  4 | d  
(3 строки)
```

Часть 3. Добавление таблицы в публикацию

1) Создайте еще одну таблицу для репликации:

```
psql -c "CREATE TABLE t1 AS SELECT * FROM t"
SELECT 3
```

```
psql -c "ALTER TABLE t1 ADD CONSTRAINT t1_key PRIMARY KEY (id)"
ALTER TABLE
```

```
psql -c "\d t1"
```

```

          Таблица "public.t1"
-----+-----+-----+-----+-----+
 Столбец | Тип      | Правило сортировки | Допустимость NULL | По умолчанию
-----+-----+-----+-----+-----+
 id       | bigint   |                     | not null           |
 t        | text     |                     |                    |

```

Индексы:

```
"t1_key" PRIMARY KEY, btree (id)
```

В отличие от таблицы `t`, автоинкрементального столбца и последовательности нет.

В базе данных подписки таблица не создана, ее нужно будет создавать вручную.

2) Посмотрите список публикаций командой `psql`:

```
psql -c "\dRp"
```

```

                          Список публикаций
Имя|Владелец|Все таблицы|Добавления|Изменения|Удаления|Опустошения|Через корень
---+-----+-----+-----+-----+-----+-----+-----+
 t |postgres|f          |t          |t          |t          |t          |f
(1 строка)

```

Реплицируются `insert`, `update`, `delete`, `truncate`.

3) Добавьте в публикацию новую таблицу:

```
psql -c "ALTER PUBLICATION t ADD TABLE t1"
ALTER PUBLICATION
```

Ошибок в логе кластеров не будет, так как мы не выполняли функцию

`pg_log_standby_snapshot()`. Они появятся после команды вставки строки в следующем пункте.

4) Вставьте строку в таблицу `t1`:

```
psql -c "INSERT INTO t1 VALUES (5, 'e')"
INSERT 0 1
```

В логе подписчика `replica1`:

```

ОШИБКА: целевое отношение логической репликации "public.t1" не существует
КОНТЕКСТ: processing remote data for replication origin "pg_43450" during
message type "INSERT" in transaction 17768, finished at 9/BC3D92F0
СООБЩЕНИЕ: фоновый процесс "logical replication worker" (PID 17622) завершился с
кодом выхода 1
СООБЩЕНИЕ: запускается применяющий процесс логической репликации для подписки
"sub1"

```

Ошибка о том, что `logical replication worker` не смог реплицировать вставку строки по причине **отсутствия таблицы `t1` на подписчике**.

В логе физической реплики `replica2`:

СООБЩЕНИЕ: 9/BC3CD550 has been already streamed, forwarding to 9/BC3D9240
 ОПЕРАТОР: START_REPLICATION SLOT "sub1" LOGICAL 9/BC3CD550 (proto_version '4',
 origin 'none', publication_names '"t"')
 СООБЩЕНИЕ: начинается логическое декодирование для слота "sub1"
 ПОДРОБНОСТИ: Передача транзакций, фиксируемых после 9/BC3D9240, чтение WAL с
 9/BC3D9088.
 ОПЕРАТОР: START_REPLICATION SLOT "sub1" LOGICAL 9/BC3CD550 (proto_version '4',
 origin 'none', publication_names '"t"')
 СООБЩЕНИЕ: процесс логического декодирования достиг точки согласованности в
 9/BC3D9088
 ПОДРОБНОСТИ: Больше активных транзакций нет.

5) Создайте структуру таблицы в базе-приемнике:

```
pg_dump -t t1 --schema-only --clean --if-exists | psql -p 5433
```

Периодические ошибки в лог кластеров перестали выводиться, но строк в таблице подписки пока не будет:

```
psql -p 5433 -c "select * from t1"
 id | t
----+---
(0 строк)
```

6) Вставьте строку на источнике:

```
psql -c "INSERT INTO t1 VALUES (6, 'f')"
```

INSERT 0 1

7) Проверьте, что строка на подписчике не появится:

```
psql -p 5433 -c "select * from t1"
 id | t
----+---
(0 строк)
```

8) На подписчике строки не появятся, пока подписка не обновится. При этом, по другим таблицам в подписке репликация будет идти:

```
psql -c "INSERT INTO t (t) VALUES ('e')"
```

```
psql -p 5433 -c "select * from t"
 id | t
----+---
  1 | a
  3 | b
  4 | d
  5 | e
(4 строки)
```

7) Обновите на подписчике подписку:

```
psql -p 5433 -c "ALTER SUBSCRIPTION sub1 REFRESH PUBLICATION"
```

ALTER SUBSCRIPTION

```
psql -p 5433 -c "select * from t1"
 id | t
----+---
(0 строк)
```

Строки пока не появились.

Через какое время появятся строки, то есть выполнится начальная синхронизация и применяются изменения?

Подписчик работает через физическую реплику. Если бы подписчик подсоединился к мастеру напрямую, то задержки бы не было.

Либо после контрольной точки на мастере, либо после вызова на источнике функции `pg_log_standby_snapshot()`.

8) Вызовите эту функцию на источнике:

```
psql -c "select pg_log_standby_snapshot()"
pg_log_standby_snapshot
-----
9/BC3D9938
(1 строка)
```

9) Проверьте, что строки на подписчике появились:

```
psql -p 5433 -c "select * from t1"
id | t
----+--
 1 | a
 2 | b
 4 | d
 5 | e
 6 | f
(5 строк)
```

В логе подписчика:

```
13:21:11.024 MSK [29400] СООБЩЕНИЕ: процесс синхронизации таблицы при логической
репликации для подписки "sub1", таблицы "t1" запущен
13:22:14.121 MSK [29400] СООБЩЕНИЕ: процесс синхронизации таблицы при логической
репликации для подписки "sub1", таблицы "t1" закончил обработку
```

В логе физической реплики:

```
13:22:14.101 MSK [29401] СООБЩЕНИЕ: процесс логического декодирования достиг точки согласованности в 9/BC3D98F0
13:22:14.101 MSK [29401] ПОДРОБНОСТИ: Больше активных транзакций нет.
13:22:14.101 MSK [29401] ОПЕРАТОР: CREATE_REPLICATION_SLOT "pg_43450_sync_43451_7353194261070147214" LOGICAL
pgoutput (SNAPSHOT 'use')
13:22:14.118 MSK [29401] СООБЩЕНИЕ: начинается логическое декодирование для слота
"pg_43450_sync_43451_7353194261070147214"
13:22:14.118 MSK [29401] ПОДРОБНОСТИ: Передача транзакций, фиксируемых после 9/BC3D9938, чтение WAL с
9/BC3D98F0.
13:22:14.118 MSK [29401] ОПЕРАТОР: START_REPLICATION_SLOT "pg_43450_sync_43451_7353194261070147214" LOGICAL
9/BC3D9938 (proto_version '4', origin 'none', publication_names 't')
```

table synchronization worker синхронизировал (передал строки) таблицу на приемнике с таблицей на источнике.

Важным является то, что при добавлении таблицы в публикацию начинается захват изменений, и после обновления подписки будет по умолчанию "бесшовно" (не блокируя доступ к таблице на источнике) выполнена синхронизация строк таблиц.

10) Очистите строки в таблице `t1` на источнике:

```
psql -c "TRUNCATE t1"
TRUNCATE TABLE
```

Часть 4. Двухнаправленная репликация

1) Создайте публикацию для таблиц `t`, `t1`:

```
psql -p 5433 -c "CREATE PUBLICATION t for TABLE t, t1;"
```

2) Создайте подписку. Имя подписки определяет по умолчанию имя логического слота репликации и должно быть уникальным во всей конфигурации. Используйте имя `sub2`.

Нельзя чтобы слот копировал данные, так как таблицы синхронизированы, поэтому нужно установить `copy_data=off`.

Нельзя допускать заикливания, поэтому `origin=none`:

```
psql -p 5432 -c "CREATE SUBSCRIPTION sub2 CONNECTION 'dbname=postgres port=5433 user=postgres' PUBLICATION t WITH (origin=none, copy_data=off)"
```

ЗАМЕЧАНИЕ: на сервере публикации создан слот репликации "sub2"

```
CREATE SUBSCRIPTION
```

В логе на 5432:

```
13:37:12.419 MSK [3882] СООБЩЕНИЕ: запускается применяющий процесс логической репликации для подписки "sub2"
```

В логе на 5433:

```
13:37:12.410 MSK [3881] СООБЩЕНИЕ: процесс логического декодирования достиг точки согласованности в 9/BC3E3E88
13:37:12.410 MSK [3881] ПОДРОБНОСТИ: Больше активных транзакций нет.
13:37:12.410 MSK [3881] ОПЕРАТОР: CREATE_REPLICATION_SLOT "sub2" LOGICAL
pgoutput (SNAPSHOT 'nothing')
13:37:12.424 MSK [3883] СООБЩЕНИЕ: начинается логическое декодирование для слота "sub2"
13:37:12.424 MSK [3883] ПОДРОБНОСТИ: Передача транзакций, фиксируемых после 9/BC3E3ED0, чтение WAL с 9/BC3E3E88.
13:37:12.424 MSK [3883] ОПЕРАТОР: START_REPLICATION SLOT "sub2" LOGICAL 0/0 (proto_version '4', origin 'none', publication_names '"t"')
13:37:12.424 MSK [3883] СООБЩЕНИЕ: процесс логического декодирования достиг точки согласованности в 9/BC3E3E88
13:37:12.424 MSK [3883] ПОДРОБНОСТИ: Больше активных транзакций нет.
13:37:12.424 MSK [3883] ОПЕРАТОР: START_REPLICATION SLOT "sub2" LOGICAL 0/0 (proto_version '4', origin 'none', publication_names '"t"')
```

Команда создания не подвиснет, так как источник и подписчик находятся в разных кластерах и подписка подсоединяется к мастеру, а не к физической реплике.

Подвисание произойдет, если базы данных источника и подписчика находятся в одном кластере.

Для продолжения работы команды нужно было бы вызывать на источнике (5433) функцию `pg_log_standby_snapshot()`.

4) Проверьте, что репликация идёт в только что созданном направлении:

```
psql -p 5433 -c "INSERT INTO t (t) VALUES ('f')"
```

ОШИБКА: повторяющееся значение ключа нарушает ограничение уникальности "t_pkey"
ПОДРОБНОСТИ: Ключ "(id)=(1)" уже существует.

Возникла ошибка. Причина - использование последовательности для генерации значений в столбце первичного ключа. Состояния последовательностей не реплицируются, и на 5433 последовательность сгенерировала значение **1**.

4) Посмотрите какие значения выдают две последовательности в двух базах данных:

```
psql -p 5433 -c "select nextval('t_id_seq')"
```

nextval
2

(1 row)

```
psql -p 5432 -c "select nextval('t_id_seq')"
```

nextval
6

(1 row)

5) Проверьте, какое максимальное значение есть в столбце реплицируемой таблицы:

```
psql -p 5433 -c "select max(id) from t"
```

max
5

(1 строка)

6) Для устранения проблемы установим, чтобы последовательность на одной базе выдавала четные числа, на другой - нечётные. В случае использования трёх баз данных, связанных репликацией, было бы три последовательности и тогда бы использовали на каждой из них INCREMENT BY 3 и отличающиеся на единицу RESTART WITH.

Переустановите значения последовательностей, чтобы они генерировали четные и нечетные числа:

```
psql -p 5432 -c "ALTER SEQUENCE t_id_seq INCREMENT BY 2 RESTART WITH 8"
psql -p 5433 -c "ALTER SEQUENCE t_id_seq INCREMENT BY 2 RESTART WITH 9"
```

Последовательности будут генерировать числа: 8, 10, 12... и 9, 11, 13...

7) Проверьте, что вставка работает:

```
psql -p 5433 -c "INSERT INTO t (t) VALUES ('g')"
```

```
psql -p 5432 -c "INSERT INTO t (t) VALUES ('h')"
```

8) Проверьте, что вставленные строки реплицировались:

```
postgres@tantor:~$ psql -p 5433 -c "select * from t"
```

id	t
1	a
2	b
4	d
5	e
9	g
8	h

(6 строк)

```
postgres@tantor:~$ psql -p 5432 -c "select * from t"
```

id	t
1	a

```

2 | b
4 | d
5 | e
9 | g
8 | h
(6 строк)
    
```

9) Проверьте, что обновления тоже работают и реплицируются:

```

psql -p 5432 -c "update t set t='HH' where id =8"
psql -p 5433 -c "update t set t='GG' where id =9"
psql -p 5432 -c "select * from t"
psql -p 5433 -c "select * from t"
    
```

```

postgres@tantor:~$ psql -p 5432 -c "select * from t"
 id | t
----+----
  1 | a
  2 | b
  4 | d
  5 | e
  8 | HH
  9 | GG
(6 строк)
    
```

```

postgres@tantor:~$ psql -p 5433 -c "select * from t"
 id | t
----+----
  1 | a
  2 | b
  4 | d
  5 | e
  8 | HH
  9 | GG
(6 строк)
    
```

Мы настроили двунаправленную репликацию. В одном направлении используется физическая реплика. Физические реплики могут использоваться в обоих направлениях.

Часть 5. Удаление подписок и публикаций

1) Удалите подписки, публикации, таблицы:

```
psql -p 5432 -c "drop subscription sub2"
psql -p 5433 -c "drop publication t"
psql -p 5433 -c "drop subscription sub1"
psql -p 5432 -c "drop publication t"
psql -p 5432 -c "checkpoint"
psql -p 5432 -c "drop table t"
psql -p 5432 -c "drop table t1"
psql -p 5433 -c "drop table t"
psql -p 5433 -c "drop table t1"
```

Примечание 1:

Если удалить слот репликации до удаления подписки, например, выдав команду:

```
psql -p 5434 -c "select pg_drop_replication_slot('sub1')"
```

то при попытке удаления подписки выдаться ошибка и подписка не будет удалена:

```
psql -p 5433 -c "drop subscription sub1"
```

ОШИБКА: слот репликации "sub1" на сервере публикации не был удалён:

ОШИБКА: слот репликации "sub1" не существует

В этом случае для удаления слота используется последовательность команд:

```
psql -p 5433 -c "alter subscription sub1 disable"
psql -p 5433 -c "alter subscription sub1 set (slot_name=none)"
psql -p 5433 -c "drop subscription sub1"
```

Примечание 2:

В процессе добавления таблиц в публикацию на физической реплике или изменения свойств подписок возможно возникновение ошибки вида:

СООБЩЕНИЕ: запускается применяющий процесс логической репликации для подписки "sub1"

ОШИБКА: не удалось начать трансляцию WAL: ОШИБКА: из слота репликации "sub1" больше нельзя получать изменения

ПОДРОБНОСТИ: Этот слот был аннулирован из-за конфликта с восстановлением.

СООБЩЕНИЕ: фоновый процесс "logical replication worker" (PID 31049) завершился с кодом выхода 1

На английском языке:

DETAIL: This slot has been invalidated because it was conflicting with recovery.

Ошибка возникает в случае:

1) `hot_standby_feedback = off` на том кластере, где создан слот логической репликации

2) `hot_standby_feedback = on`, но при этом отсутствует физический слот репликации на мастере для физической реплики, на которой создан слот логической репликации.

Причина: автовакуум на мастере удаляет старые версии строк из таблиц системного каталога, которые нужны для логического декодирования на том кластере, где создан слот логической репликации.

Описание:

<https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=6af1793954e8c5e753af83c3edb37ed3267dd179>

Глава 10. Новые возможности Tantor Postgres 17

Расширение orafce

1) Посмотрите, какие расширения установлены в базе:

```
postgres=# \dx
```

```

                                List of installed extensions
  Name          | Version | Schema  | Description
-----+-----+-----+-----
hypopg         | 1.4.1   | public  | Hypothetical indexes for PostgreSQL
pg_columnar   | 11.1-12 | public  | Hydra Columnar extension
pg_stat_statements | 1.11   | public  | track planning and execution
statistics of
pg_store_plans | 1.8.1   | public  | track plan statistics of all SQL
statements
plpgsql       | 1.0     | pg_catalog | PL/pgSQL procedural language
plpython3u    | 1.0     | pg_catalog | PL/Python3U untrusted procedural
language
(6 rows)

```

Список в вашей базе может отличаться от приведенного.

2) Проверьте, доступно ли для установки расширение orafce:

```
postgres=# select * from pg_available_extensions where name ilike '%ora%';
```

```

 name | default_version | installed_version | comment
-----+-----+-----+-----
 orafce | 4.13            | 4.13             | Functions and operators that
 emulate a subset of functions a
(1 строка)

```

3) Какие схемы есть в базе? Получите список схем:

```
postgres=# \dn
```

```

          Список схем
  Имя   | Владелец
-----+-----
 public | pg_database_owner
(1 строка)

```

4) Установите в базу расширение orafce:

```
postgres=# CREATE EXTENSION orafce;
CREATE EXTENSION
```

5) Получите список схем:

```
postgres=# \dn
```

```

          Список схем
  Имя   | Владелец
-----+-----
 dbms_alert   | postgres
 dbms_assert  | postgres
 dbms_output  | postgres
 dbms_pipe    | postgres
 dbms_random  | postgres
 dbms_sql     | postgres
 dbms_utility | postgres
 oracle       | postgres
 plunit       | postgres
 plvchr       | postgres
 plvdate      | postgres

```

```

plvlex      | postgres
plvstr     | postgres
plvsubst   | postgres
public     | pg_database_owner
utl_file   | postgres
(16 строк)

```

Расширение создало 15 схем.

В Oracle Database есть объекты - пакеты процедур. В PostgreSQL пакетов нет. Пакеты используются для объединения подпрограмм. Близкий аналог пакетов - схемы. В отличие от пакетов в схемах могут находиться объекты любого типа, а не только подпрограммы.

В Oracle Database пакеты поставляемые стандартно имеют префикс "dbms_"

6) Часть объектов, которые в Oracle Database вызываются без префикса имени пакета, создаются расширением в схеме oracle. Вставьте имя этой схемы в путь поиска:

```

postgres=# set search_path TO "$user", public, oracle;
SET

```

7) Обратитесь к таблице dual, используемой приложениями, работающими с Oracle Database для вызова однострочных функций. В Oracle Database обязательно использование фразы FROM в команде SELECT, в PostgreSQL - необязательно. Приложения в Oracle Database обычно используют команду "SELECT функция() FROM DUAL;".

Выполните команду:

```

postgres=# SELECT sysdate() FROM dual;

```

Можно заметить, что круглые скобки обязательны. В Oracle Database функция SYSDATE используется без круглых скобок. В PostgreSQL функции без аргументов не могут вызываться без круглых скобок, за исключением тех, которые по стандарту SQL вызываются без круглых скобок. Например: current_date, current_timestamp, current_catalog, current_role, current_user, session_user, user, current_schema. Причем, из этих функций только current_schema может вызываться с круглыми скобками.

8) Расширение создаёт используемый в Oracle Database тип данных VARCHAR2:

```

postgres=# select 'hello'::varchar2;
 varchar2
-----
hello
(1 строка)

```

9) Расширение создаёт функции, которые используются в Oracle Database для отладочного вывода в составе пакета процедур dbms_output:

```

postgres=# SELECT dbms_output.serveroutput(true);
serveroutput
-----

(1 строка)

```

Аналог команды в Oracle Database "SET SERVEROUTPUT ON"

```

postgres=# SELECT dbms_output.put('aa');
put
-----

```

(1 строка)

```
postgres=# SELECT dbms_output.put('bb');
put
-----
```

(1 строка)

```
postgres=# SELECT * FROM dbms_output.get_lines(1);
lines | numlines
-----+-----
{aabb} |          1
```

(1 строка)

```
postgres=# SELECT dbms_output.put('aa');
put
-----
```

(1 строка)

```
postgres=# SELECT dbms_output.put('bb');
put
-----
```

(1 строка)

```
postgres=# SELECT * FROM dbms_output.get_line();
line | status
-----+-----
aabb |          0
```

(1 строка)

Результат `get_line()` и `get_lines(1)` одинаков.

Результат `enable()` и `serveroutput(true)` одинаков.

10) Верните значение параметра пути поиска к значению по умолчанию:

```
postgres=# reset search_path;
RESET
```

Расширение pg_variables

1) Расширение позволяет использовать переменные для хранения значений на уровне сессии. Расширение обеспечивает функционал, аналогичный переменным пакетов процедур в Oracle Database. Функционал также схож с атрибутами "application contexts" в Oracle Database.

Создание переменных

Преимущество использования переменных: быстрота доступа. Переменные могут использоваться как более производительная и простая альтернатива временным таблицам.

Установите расширение:

```
postgres=# CREATE EXTENSION pg_variables;
CREATE EXTENSION
```

2) Установите значение 101 для переменной ("атрибуту") int1 в "пакете" ("контексте", группе переменных) с названием vars. Термин "пакет" используется в расширении для обозначения групп переменных.

```
postgres=# SELECT pgv_set('vars', 'int1', 101);
   pgv_set
-----
```

(1 строка)

3) Установите [текстовую](#) переменную в том же пакете:

```
postgres=# SELECT pgv_set('vars', 'text1', 'text variable'::text, true);
   pgv_set
-----
```

(1 строка)

4) Для получения значений используется функция `pgv_get`. Первый и второй параметры понятны: имя пакета и переменной. Третий аргумент - тип переменной. Выполните команду и посмотрите результат:

```
postgres=# SELECT pgv_get('vars', 'int1');
ERROR:  function pgv_get(unknown, unknown) does not exist
```

Ошибка означает, что значения по умолчанию у третьего параметра функции нет.

5) Пустое значение не передаётся:

```
postgres=# SELECT pgv_get('vars', 'int1', null);
ERROR:  function pgv_get(unknown, unknown, unknown) is not unique
```

6) Пакет знает тип переменной и сообщает его:

```
postgres=# SELECT pgv_get('vars', 'int1', null::numeric);
ERROR:  variable "int1" requires "integer" value
```

7) Передаём значение этого типа - функция возвращает значение:

```
postgres=# SELECT pgv_get('vars', 'int1', 0);
   pgv_get
-----
      101
```

(1 строка)

8) Можно использовать и пустое значение `NULL::int` заданного типа:

```
postgres=# SELECT pgv_get('vars', 'int1', NULL::int);
pgv_get
-----
```

101

(1 строка)

9) Создать две переменные с одинаковым названием, но разных типов - нельзя:

```
postgres=# SELECT pgv_set('vars', 'int1', null::text);
ERROR: variable "int1" requires "integer" value
```

10) Получение значения текстовой переменной:

```
postgres=# SELECT pgv_get('vars', 'text1', NULL::text);
pgv_get
-----
```

text variable

(1 строка)

11) Список переменных:

```
postgres=# SELECT * FROM pgv_list() order by package, name;
```

package	name	is_transactional
vars	int1	f
vars	text1	f

(2 строки)

По умолчанию `is_transactional=false` и на а работу с переменными не влияет, открыта транзакция или нет. Если `is_transactional=true`, то при откате транзакции, в том числе до точек сохранения, действия с переменными будут откатываться.

12) Транзакционность переменной задаётся четвертым параметром функции `pgv_set` в момент создания переменной. Переопределить его после создания нельзя:

```
postgres=# SELECT pgv_set('vars', 'text1', 'text variable'::text, true);
ERROR: variable "text1" already created as NOT TRANSACTIONAL
```

13) Можно удалить переменную и создать с тем же именем заново:

```
postgres=# SELECT pgv_remove('vars', 'text1');
pgv_remove
-----
```

(1 строка)

```
postgres=# SELECT pgv_set('vars', 'text1', 'text variable'::text, true);
pgv_set
-----
```

(1 строка)

```
postgres=# SELECT * FROM pgv_list() order by package, name;
```

package	name	is_transactional
vars	int1	f
vars	text1	t

(2 строки)

14) Использование транзакционных переменных не приводит к накрутке счетчика транзакций. Проверим это. Текущий номер транзакции в кластере:

```
postgres=# SELECT pg_current_xact_id();
 pg_current_xact_id
-----
                871
```

(1 строка)

15) Откроем транзакцию, создадим транзакционную переменную и зафиксируем транзакцию:

```
postgres=# begin transaction;
BEGIN
postgres=# SELECT pgv_set('vars', 'text2', 'text variable'::text, true);
 pgv_set
-----
```

(1 строка)

```
postgres=# SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
-----
```

(1 строка)

Номер транзакции не назначен, используется виртуальный номер.

16) После фиксации транзакции функция получения номера транзакции выдаёт следующий номер:

```
postgres=# commit;
COMMIT
postgres=# SELECT pg_current_xact_id();
 pg_current_xact_id
-----
                872
```

(1 строка)

```
postgres=# SELECT pg_current_xact_id();
 pg_current_xact_id
-----
                873
```

(1 строка)

Это значит, что транзакция, в которой была создана транзакционная переменная, не использовала реальный номер транзакции.

Получение реального номера транзакции вносило бы задержку. Работа с транзакционными переменными так же эффективна, как с нетранзакционными.

17) Используемая память по пакетам:

```
postgres=# SELECT * FROM pgv_stats() order by package;
 package | allocated_memory
-----+-----
    vars  |              16384
```

(1 строка)

18) Удаление переменной с названием int1:

```
postgres=# SELECT pgv_remove('vars', 'int1');
pgv_remove
-----
```

(1 строка)

19) Удаление пакета с переменными этого пакета:

```
postgres=# SELECT pgv_remove('vars');
pgv_remove
-----
```

(1 строка)

20) Удаление всех пакетов и всех переменных:

```
postgres=# SELECT pgv_free();
pgv_free
-----
```

(1 строка)

В любом случае, срок жизни переменных - до окончания сессии.

Расширение `page_repair`

Часть 1. Подготовка реплики

Расширение `page_repair` включает в себя разделяемую библиотеку и две функции. Функции позволяют скопировать через сетевое соединение с физической реплики один блок на один вызов процедуры.

Для использования расширения нужна физическая реплика. Если она есть, то можно пропустить пункты её создания. Создание физической реплики рассматривалось в практике 8а.

Остановка кластера, присутствующего в виртуальной машине:

```
postgres@tantor:~$ sudo systemctl stop tantor-se-server-17-replica
```

Остановка кластера, если он был создан и не пригоден к использованию как физическая реплика (стал мастером, не может накладывать журнальные данные из-за отсутствия файлов журналов):

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17-replica/data1
postgres@tantor:~$ rm -rf /var/lib/postgresql/tantor-se-17-replica/data1
```

Создание реплики:

```
postgres@tantor:~$ rm /var/lib/postgresql/tantor-se-17/data/global/pg_store_plans.stat
postgres@tantor:~$ pg_basebackup -D /var/lib/postgresql/tantor-se-17-replica/data1 -P -R -C --slot=replica1 --checkpoint=fast
```

Если резервирование прервать, то нужно будет удалить директорию: `rm -rf /var/lib/postgresql/tantor-se-17-replica/data1`

и слот на мастере: `psql -c "select pg_drop_replication_slot('replica1')"`

```
postgres@tantor:~$ echo "port=5433" >> /var/lib/postgresql/tantor-se-17-replica/data1/postgresql.auto.conf
```

Запуск реплики:

```
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1 -l log_replica1.log
```

Проверка, что репликация работает:

```
postgres@tantor:~$ psql -c "select * from pg_replication_slots"
```

В столбце `status` должно быть значение `"t"`.

Часть 2. Подготовка таблицы

1) Создайте таблицу и заполните её данными:

```
postgres=# drop table if exists t;
NOTICE: table "t" does not exist, skipping
DROP TABLE
postgres=# CREATE TABLE t (id bigserial primary key, t text);
CREATE TABLE
postgres=# INSERT INTO t(t) SELECT encode((floor(random()*1000)::numeric ^
100)::numeric)::text::bytea, 'base64') from generate_series(1,1000);
INSERT 0 1000
postgres=# update t set t = t || 'a';
UPDATE 1000
```

Была вставлена тысяча строк и обновлена тысяча строк. В страницах имеются актуальные и неактуальные версии строк, пока не отработает автовакуум.

2) Размер файла таблицы:

```
postgres=# select pg_relation_size('t');
pg_relation_size
-----
            802816
(1 строка)
```

3) Относительный путь к файлу со строками:

```
postgres=# SELECT pg_relation_filepath('t'::regclass);
pg_relation_filepath
-----
base/5/16622
(1 строка)
```

4) Префикс для получения абсолютного пути из относительного (он же PGDATA):

```
postgres=# \dconfig data_directory
                Список параметров конфигурации
Параметр      | Значение
-----+-----
data_directory | /var/lib/postgresql/tantor-se-17/data
(1 строка)
```

5) **Номер блока**, в котором расположена строка с id=900:

```
postgres=# select ctid, id from t where id=900;

 ctid | id
-----+-----
(92,15) | 900
(1 строка)
```

6) Остановите экземпляр:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data
```

ожидание завершения работы сервера.... готово
сервер остановлен

7) Вставьте мусор в блок, в котором содержится строка с id=100:

```
postgres@tantor:~$ dd if=/dev/urandom conv=notrunc bs=8192 seek=92 count=1
of=/var/lib/postgresql/tantor-se-17/data/base/5/16622
```

1+0 записей получено
 1+0 записей отправлено
 8192 байт (8.2 kB, 8.0 KiB) скопирован, 0.000300021 s, 27.3 MB/s

8) Запустите кластер:

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
```

9) Выполните команды, которые требуют обращения к поврежденной странице:

```
postgres=# select ctid, id from t where id=900;
ERROR:  invalid page in block 92 of relation base/5/16622
postgres=# select count(*) from t;
ERROR:  invalid page in block 92 of relation base/5/16622
postgres=# analyze verbose t;
INFO:  analyzing "public.t"
ERROR:  invalid page in block 92 of relation base/5/16622
```

10) Заморозка не может быть выполнена:

```
postgres=# select pg_current_xact_id();
 pg_current_xact_id
-----
                797
```

(1 строка)

```
postgres=# vacuum freeze t;
ERROR:  invalid page in block 92 of relation base/5/16622
КОНТЕКСТ:  while scanning block 92 of relation "public.t"
```

```
postgres=# select relfrozenxid from pg_class where relname='t';
 relfrozenxid
-----
                795
```

(1 строка)

11) Команды с **полным сканированием** таблицы, дойдя до сбойного блока, также прервут работу:

```
postgres=# explain update t set t = t || 'b' where id > 100;
               QUERY PLAN
-----
 Update on t  (cost=0.00..112.75 rows=0 width=0)
   ->  Seq Scan on t  (cost=0.00..112.75 rows=900 width=38)
         Filter: (id > 100)
```

(3 строки)

```
postgres=# explain (analyze) update t set t = t || 'b' where id > 100;
ERROR:  invalid page in block 54 of relation base/5/16622
```

12) Команды с индексным доступом, не считывающие сбойный блок, могут выполняться:

```
postgres=# update t set t = t || 'b' where id<500;
UPDATE 499
```

13) Вакуумирование, если обратится к поврежденному блоку (определяется по карте видимости), не может выполняться. Старые версии строк не будут очищаться, файлы таблиц будут увеличиваться в размерах.

```
postgres=# vacuum verbose t;
INFO:  vacuuming "postgres.public.t"
ERROR:  invalid page in block 92 of relation base/5/16622
КОНТЕКСТ:  while scanning block 92 of relation "public.t"
```


Часть 3. Восстановление страницы с помощью `page_repair`

1) Установите расширение в базу данных с таблицей, в которой есть повреждённая страница:

```
postgres=# CREATE EXTENSION page_repair;
CREATE EXTENSION
```

2) Посмотрите определения двух функций, входящих в расширение:

```
postgres=# \df pg_repair_page
```

Схема	Имя	Тип данных результата	Список функций	Типы данных аргументов	Тип
public	pg_repair_page	boolean		regclass, bigint, text	функ.
public	pg_repair_page	boolean		regclass, bigint, text, text	функ.

(2 строки)

3) Вызовите функцию для восстановления страницы:

```
postgres=# select pg_repair_page('t'::regclass, 92, 'port=5433');
ERROR: data checksums are not enabled
```

Расширению нужны контрольные суммы, включённые на кластере. Контрольные суммы нужны, чтобы отказать в восстановлении, если администратор захочет восстановить неповреждённый блок. Исследовать содержимое блока сложно, а по контрольной сумме просто.

4) Включите подсчет контрольных сумм на кластере с поврежденной таблицей:

```
postgres=# \q
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data
ожидание завершения работы сервера.... готово
сервер остановлен
postgres@tantor:~$ rm /var/lib/postgresql/tantor-se-17/data/global/pg_store_plans.stat
postgres@tantor:~$ pg_checksums -e -D /var/lib/postgresql/tantor-se-17/data
Обработка контрольных сумм завершена
Просканировано файлов: 1271
Просканировано блоков: 27179
Записано файлов: 1055
Записано блоков: 27178
pg_checksums: синхронизация каталога данных
pg_checksums: модификация управляющего файла
Контрольные суммы в кластере включены
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
```

5) Вызовите ещё раз функцию для восстановления страницы:

```
postgres@tantor:~$ psql -c "select pg_repair_page('t'::regclass, 92, 'port=5433')"
NOTICE: skipping page repair of the given page --- page is not corrupted
pg_repair_page
-----
t
(1 строка)
```

Функция успешно выполнялась, сообщив что не стала восстанавливать страницу, так как по ее логике страница не повреждена.

6) Проверьте, повреждена ли страница:

```
postgres@tantor:~$ psql -c "select ctid, id from t where id=900"
ERROR: invalid page in block 92 of relation base/5/16622
```

Страница всё так же повреждена. Почему расширение считает страницу неповрежденной?

При включении подсчета контрольных сумм, они были рассчитаны и для поврежденного блока. Утилита включения не может проверять блоки на логическом уровне, она рассчитывает контрольные суммы и вставляет их в блоки.

7) Чтобы затереть контрольную сумму, повторите процедуру повреждения страницы:

```
pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data
dd if=/dev/urandom conv=notrunc bs=8192 seek=92 count=1
of=/var/lib/postgresql/tantor-se-17/data/base/5/16622
sudo systemctl start tantor-se-server-17
```

8) Повторите процедуру восстановления страницы:

```
postgres@tantor:~$ psql -c "select pg_repair_page('t'::regclass, 92,
'port=5433')"
ERROR: page on standby is also corrupted
```

Функция сообщает, что по её логике страница на реплике тоже повреждена.

9) Проверьте, повреждена ли страница на реплике:

```
postgres@tantor:~$ psql -p 5433 -c "select ctid, id from t where id=900"
 ctid  | id
-----+-----
(92,15) | 900
(1 строка)

postgres@tantor:~$ psql -p 5433 -c "select count(*) from t"
 count
-----
  1000
(1 строка)
```

Страницы таблицы на реплике не повреждены. Почему расширение отказывается восстанавливать страницу?

Потому, что на реплике не включён подсчет контрольных сумм. Текст ошибки вводит в заблуждение.

10) Включите подсчёт контрольных сумм блоков данных на реплике:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-17-replica/data1
ожидание завершения работы сервера.... готово
сервер остановлен
postgres@tantor:~$ rm /var/lib/postgresql/tantor-se-17-
replica/data1/global/pg_store_plans.stat
postgres@tantor:~$ pg_checksums -e -D /var/lib/postgresql/tantor-se-17-
replica/data1
Обработка контрольных сумм завершена
Просканировано файлов: 1271
Просканировано блоков: 27179
Записано файлов: 1055
Записано блоков: 27178
pg_checksums: синхронизация каталога данных
pg_checksums: модификация управляющего файла
Контрольные суммы в кластере включены
postgres@tantor:~$ pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1
-l log_replica1.log
```

11) Повторите процедуру восстановления страницы:

```

postgres@tantor:~$ psql -c "select pg_repair_page('t'::regclass, 92,
'port=5433')"
 pg_repair_page
-----
 t
(1 строка)
  
```

12) Проверьте, читаются ли страницы таблицы:

```

postgres@tantor:~$ psql -c "select count(*) from t"
 count
-----
 1000
(1 строка)
  
```

Страницы читаются, страница была восстановлена путём копирования с физической реплики.

Использование расширения `page_repair` требует включенного подсчета контрольных сумм на мастере и физической реплике, с которой страница будет копироваться на мастер.

Включение подсчета контрольных сумм вставляет контрольную сумму в любые блоки, в том числе повреждённые.

Часть 4. Обнуление страницы

В отсутствие физических реплик и/или возможности восстановиться с бэкапов, восстановить повреждённый блок нельзя. Оставлять такой блок в таблице тоже нельзя - вакуумирование и заморозка не будут обрабатывать. Есть возможность сделать сбойную страницу пустой. При этом всё содержимое блока считается не существующим.

1) Повторите процедуру повреждения блока:

```
pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data
dd if=/dev/urandom conv=notrunc bs=8192 seek=92 count=1
of=/var/lib/postgresql/tantor-se-17/data/base/5/16622
sudo systemctl start tantor-se-server-17
psql -c "select ctid, id from t where id=900"
WARNING: page verification failed, calculated checksum 9494 but expected 37021
ERROR: invalid page in block 92 of relation base/5/16622
```

При включённых контрольных суммах к ошибке добавилось [предупреждение](#).

2) Включите на уровне сессии параметр:

```
postgres=# set zero_damaged_pages = on;
SET
```

3) Выполните запрос к таблице:

```
postgres=# select count(*) from t;
 count
-----
  1000
(1 строка)
```

Число строк верное, ошибок нет. Почему?

Потому что автовакуум обработал таблицу, обновил карту видимости, во всех блоках только актуальные версии строк. Поэтому при использовании [сканирования только индекса](#), серверному процессу не нужно считывать блоки таблицы для проверки, актуальна ли строка, на которую ссылается индексная запись.

```
postgres=# explain select count(*) from t;
          QUERY PLAN
-----
Aggregate  (cost=49.77..49.78 rows=1 width=8)
->  Index Only Scan using t_pkey on t  (cost=0.28..47.27 rows=1000 width=0)
(2 строки)
```

4) Выполните команду:

```
postgres=# select count(*) from t where t is not null;

WARNING: page verification failed, calculated checksum 9494 but expected 37021
WARNING: invalid page in block 92 of relation base/5/16622; zeroing out page
 count
-----
   980
(1 строка)
```

Число строк другое - на 20 строк меньше. В повреждённом блоке находилось 20 строк, они считаются исчезнувшими.

Предупреждающие сообщения - это результат установки параметра `zero_damaged_pages = on` и включенного подсчёта контрольных сумм. Если бы контрольные суммы были отключены, то предупреждений бы не было, но результат (980) был бы тот же.

5) Выполните команду:

```
postgres=# vacuum freeze t;
VACUUM
```

Вакуумирование проходит успешно, считая блок пустым.

При этом блок не менялся и не будет меняться в файле. Параметр `zero_damaged_pages = on` не меняет содержимое блока в файле.

```
pg_ctl stop -D /var/lib/postgresql/tantor-se-17/data
dd if=/dev/zero conv=notrunc bs=8192 seek=92 count=1
of=/var/lib/postgresql/tantor-se-17/data/base/5/16622
sudo systemctl start tantor-se-server-17
psql -c "select ctid, id from t where id=900"
 ctid | id
-----+-----
(0 строк)
```

Содержимое сбойного блока заполнено нулями. Контрольная сумма правильная - тоже нули. Блок считается неповреждённым, просто пустым.

6) Выполните команды:

```
postgres@tantor:~$ psql -c "select count(*) from t"
 count
-----
  999
(1 строка)
```

```
postgres@tantor:~$ psql -c "select ctid, id from t where id=901"
 ctid | id
-----+-----
(0 строк)
```

```
postgres@tantor:~$ psql -c "select count(*) from t"
 count
-----
  998
(1 строка)
```

Количество строк меняется в результате выборки.

Серверный процесс использует **индексное сканирование** (не `Index Only Scan`), проверяет содержимое блока и не обнаруживает строку:

```
postgres@tantor:~$ psql -c "explain select ctid, id from t where id=903"
                                QUERY PLAN
-----
Index Scan using t_pkey on t  (cost=0.28..8.29 rows=1 width=14)
  Index Cond: (id = 903)
(2 строки)
```

7) Перестройте индексы:

```
postgres=# reindex (verbose) table t;
INFO:  index "t_pkey" was reindexed
ПОДРОБНОСТИ:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

```
INFO: index "pg_toast_16622_index" was reindexed
ПОДРОБНОСТИ: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
REINDEX
postgres=# select count(*) from t;
 count
-----
    980
(1 строка)
```

8) Удалите таблицу:

```
postgres=# drop table t;
DROP TABLE
```

Отладка подпрограмм

Часть 1. Установка расширения из исходных кодов на примере pldebugger

Эта часть практики иллюстрирует установку модулей, поставляемых из исходных кодов.

Расширение, пример установки которого рассматривается, может быть полезным для разработчиков при работе с базами данных, на которых ведётся разработка.

Для отладки подпрограмм нужна поддержка со стороны сервера и графическое клиентское приложение (среда разработки), которое будет показывать исходный код подпрограммы, инициировать отладку и получать отладочную информацию. Функционал стандартный для отладчиков: устанавливать точки останова, пошаговое выполнение, наблюдение за переменными и их изменение.

Серверная часть - модуль (библиотека и расширение) создан [EnterpriseDB](https://github.com/EnterpriseDB/pldebugger), свободно распространяется, размещен по адресу <https://github.com/EnterpriseDB/pldebugger>

Основное клиентское приложение - pgAdmin. Другие клиентские приложения могут использовать серверную часть.

1) Переключитесь в root, так как он владелец программного обеспечения:

```
astra@tantor:~$ su -
Пароль: root
root@tantor:~#
```

2) Скачайте расширение **pldebugger**:

```
root@tantor:~#
wget https://github.com/EnterpriseDB/pldebugger/archive/refs/heads/master.zip
```

3) Распакуйте архив:

```
root@tantor:~# unzip master.zip
```

4) Перейдите в директорию, в которую были распакованы исходные файлы расширения:

```
root@tantor:~# cd pldebugger-master
```

5) Добавьте в путь директорию с утилитой `pg_config` и переменную окружения, указывающую утилите `make` использовать логику установки расширений PGXS:

```
root@tantor:~/pldebugger-master# export PATH=/opt/tantor/db/17/bin:$PATH
export USE_PGXS=1
```

6) В файле `README.pldebugger` написано как устанавливать расширение. Дайте первую команду:

```
root@tantor:~/pldebugger-master# make
```

Появится предупреждение:

```
plpgsql_debugger.c: In function 'is_datum_visible':
plpgsql_debugger.c:1258:36: warning: declaration of 'i' shadows a previous local
[-Wshadow=compatible-local]
1258 |             int i;
    |             ^
plpgsql_debugger.c:1234:43: note: shadowed declaration is here
```

```
1234 |                               int                               i;
      |                               |                               ^
```

показывающее качество написания кода расширения.

7) Следующая команда, описанная в файле `README.pldebugger` - это копирование файлов расширения в стандартные директории программного обеспечения.

Выполните команду:

```
root@tantor:~/pldebugger-master# make install
```

```
/usr/bin/mkdir -p '/opt/tantor/db/17/lib/postgresql'
/usr/bin/mkdir -p '/opt/tantor/db/17/share/postgresql/extension'
/usr/bin/mkdir -p '/opt/tantor/db/17/share/postgresql/extension'
/usr/bin/mkdir -p '/opt/tantor/db/17/share/doc/postgresql/extension'
/usr/bin/install -c -m 755 plugin_debugger.so '/opt/tantor/db/17/lib/postgresql/plugin_debugger.so'
/usr/bin/install -c -m 644 ./pldbgapi.control '/opt/tantor/db/17/share/postgresql/extension/'
/usr/bin/install -c -m 644 ./pldbgapi--1.1.sql ./pldbgapi--unpacked--1.1.sql ./pldbgapi--1.0--1.1.sql '/opt/tantor/db/17/share/postgresql/extension/'
/usr/bin/install -c -m 644 ./README-pldebugger.md '/opt/tantor/db/17/share/doc/postgresql/extension/'
/usr/bin/mkdir -p '/opt/tantor/db/17/lib/postgresql/bitcode/plugin_debugger'
/usr/bin/mkdir -p '/opt/tantor/db/17/lib/postgresql/bitcode'/plugin_debugger/
/usr/bin/install -c -m 644 plpgsql_debugger.bc '/opt/tantor/db/17/lib/postgresql/bitcode'/plugin_debugger/./
/usr/bin/install -c -m 644 plugin_debugger.bc '/opt/tantor/db/17/lib/postgresql/bitcode'/plugin_debugger/./
/usr/bin/install -c -m 644 dbgcomm.bc '/opt/tantor/db/17/lib/postgresql/bitcode'/plugin_debugger/./
/usr/bin/install -c -m 644 pldbgapi.bc '/opt/tantor/db/17/lib/postgresql/bitcode'/plugin_debugger/./
cd '/opt/tantor/db/17/lib/postgresql/bitcode' && /usr/lib/llvm-13/bin/llvm-lto -thinlto -thinlto-action=thinlink -o plugin_debugger.index.bc plugin_debugger/plpgsql_debugger.bc plugin_debugger/plugin_debugger.bc plugin_debugger/dbgcomm.bc plugin_debugger/pldbgapi.bc
```

8) Обновите список файлов для поиска:

```
root@tantor:~/pldebugger-master# updatedb
```

Поищите файл модуля:

```
root@tantor:~/pldebugger-master# locate plugin_debugger
/opt/tantor/db/17/lib/postgresql/plugin_debugger.so
bitcode/plugin_debugger
bitcode/plugin_debugger.index.bc
bitcode/plugin_debugger/dbgcomm.bc
bitcode/plugin_debugger/pldbgapi.bc
bitcode/plugin_debugger/plpgsql_debugger.bc
bitcode/plugin_debugger/plugin_debugger.bc
/root/pldebugger-1.5/plugin_debugger.bc
/root/pldebugger-1.5/plugin_debugger.c
/root/pldebugger-1.5/plugin_debugger.def
/root/pldebugger-1.5/plugin_debugger.o
/root/pldebugger-1.5/plugin_debugger.so
```

Этот пункт иллюстрирует один из способов быстрого поиска файлов в операционной системе.

Файл модуля был установлен в директории:

```
/opt/tantor/db/17/lib/postgresql/plugin_debugger.so
```

Название файла модуля нужно знать, чтобы загрузить библиотеку.

9) Вернитесь в терминал непривилегированного пользователя:

```
root@tantor:~/pldebugger-master# exit
logout
```

10) Проверьте, что расширение доступно для установки в базу:

```
astra@tantor:~$ psql
postgres=# select * from pg_available_extensions where name like '%dbg%';
 name | default_version | installed_version | comment
-----+-----+-----+-----
```

```
pldbgapi | 1.1 | server-side support for debugging PL/pgSQL functions
(1 строка)
```

11) Посмотрите значение параметра:

```
postgres=# \dconfig shared_preload_libraries
                Список параметров конфигурации
    Параметр      | Значение
-----+-----
shared_preload_libraries | pg_stat_statements, pg_qualstats, pg_store_plans,
                                pg_prewarm, pg_stat_kcache
```

(1 строка)

12) Добавьте библиотеку в конец списка:

```
postgres=# alter system set shared_preload_libraries = pg_stat_statements,
pg_qualstats, pg_store_plans, pg_prewarm, pg_stat_kcache, plugin_debugger ;
ALTER SYSTEM
```

Апострофы после знака равенства нельзя использовать, иначе команда добавит кавычки,

трактуя строку как название файла, и экземпляра не запустится. Пример команды, которая

выполнится, но экземпляр не запустится, пока вручную не будет отредактирован файл

postgresql.auto.conf, так как команду ALTER SYSTEM не выполняется на остановленном

экземпляре:

```
alter system set shared_preload_libraries = 'pg_stat_statements, pg_store_plans,
auto_explain, plugin_debugger';
postgres=# \q
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$ pg_ctl start
waiting for server to start....
ВАЖНО: нет доступа к файлу "pg_stat_statements, pg_qualstats, pg_store_plans,
pg_prewarm, pg_stat_kcache, plugin_debugger": Нет такого файла или каталога
СООБЩЕНИЕ: система БД выключена
stopped waiting
pg_ctl: could not start server
Examine the log output.
```

13) Рестаруйте экземпляр:

```
astra@tantor:~$ sudo systemctl restart tantor-se-server-17
```

14) Библиотека отладчика была загружена. Создайте расширение в базе данных postgres:

```
astra@tantor:~$ psql
postgres=# create extension pldbgap;
CREATE EXTENSION
```

15) Создайте функцию для тестирования отладчика:

```
CREATE OR REPLACE FUNCTION bobdef()
RETURNS text
LANGUAGE plpgsql
SECURITY DEFINER
AS $function$
BEGIN
    RAISE NOTICE 'search_path %', current_schemas(true);
    RAISE NOTICE 'current_user %', current_user;
    RAISE NOTICE 'session_user %', session_user;
    RAISE NOTICE 'user %', user;
```

```
RETURN now();  
END;  
$function$  
;
```

Часть 2. Отладка функции в pgAdmin

1) Запустите pgAdmin.

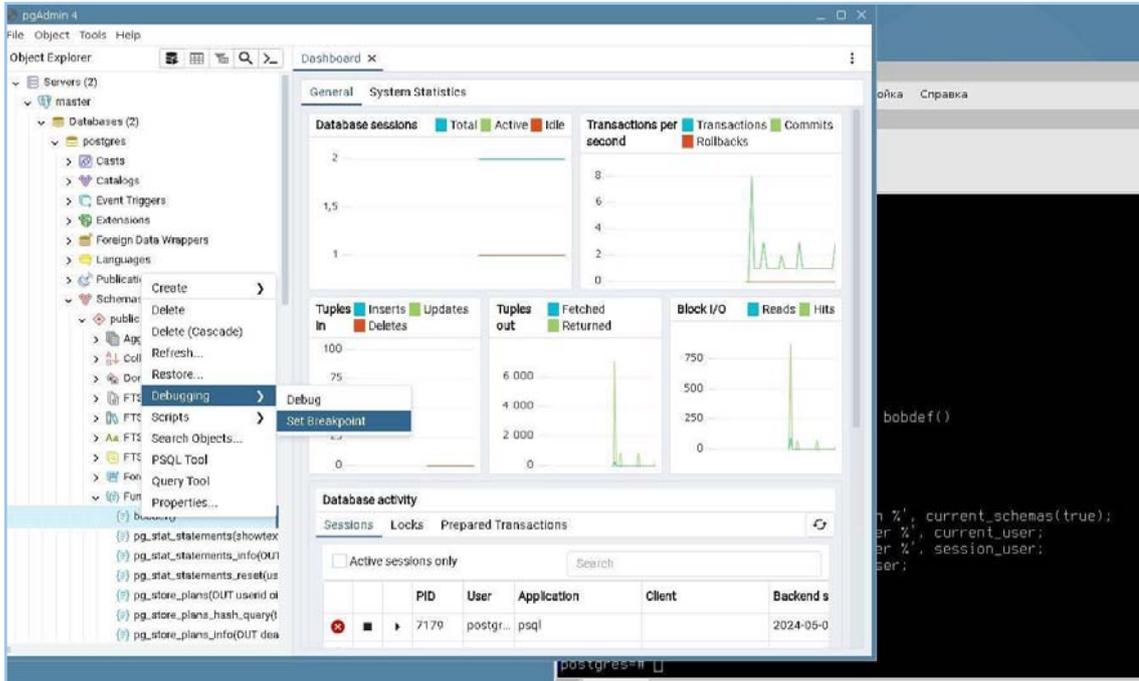
При запросе пароля наберите tantor.

2) Раскройте Servers -> master -> Schemas (1) -> public -> Functions (..)

Если соединений с базой нет, создайте его и назовите master.

3) Для отладки выполнения подпрограммы в чужой сессии, выберите функцию `bobdef()`.

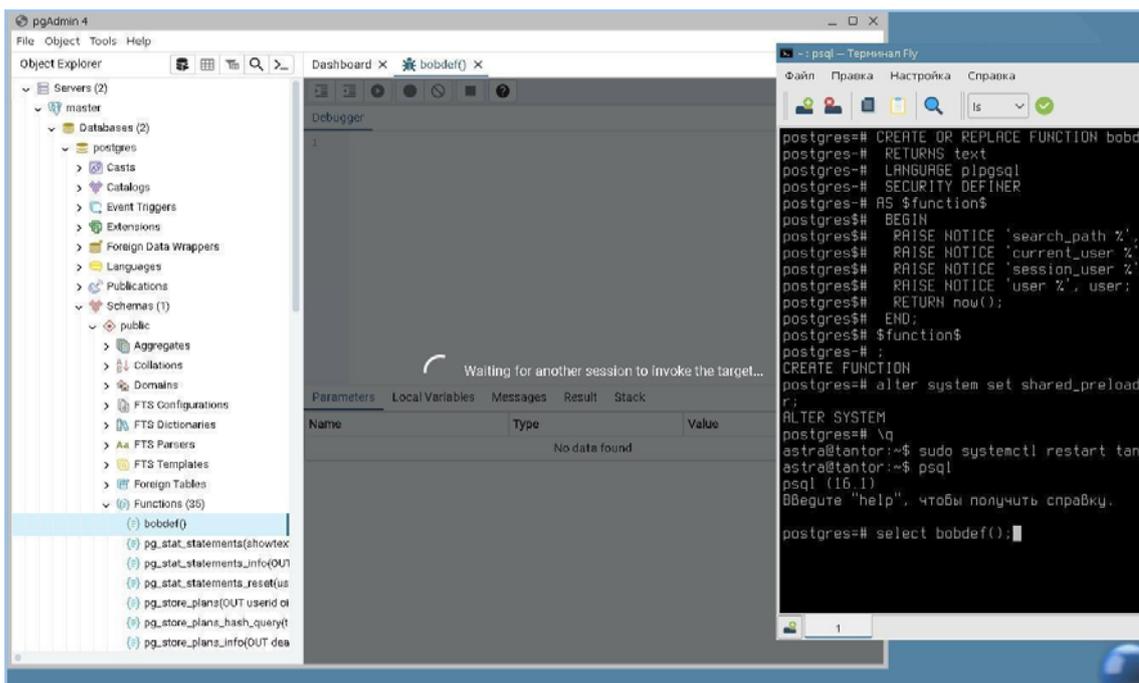
Нажмите правую кнопку мыши и выберите Debugging -> Set Breakpoint



4) Появится сообщение "Waiting for another session to invoke target".

В psql вызовите функцию:

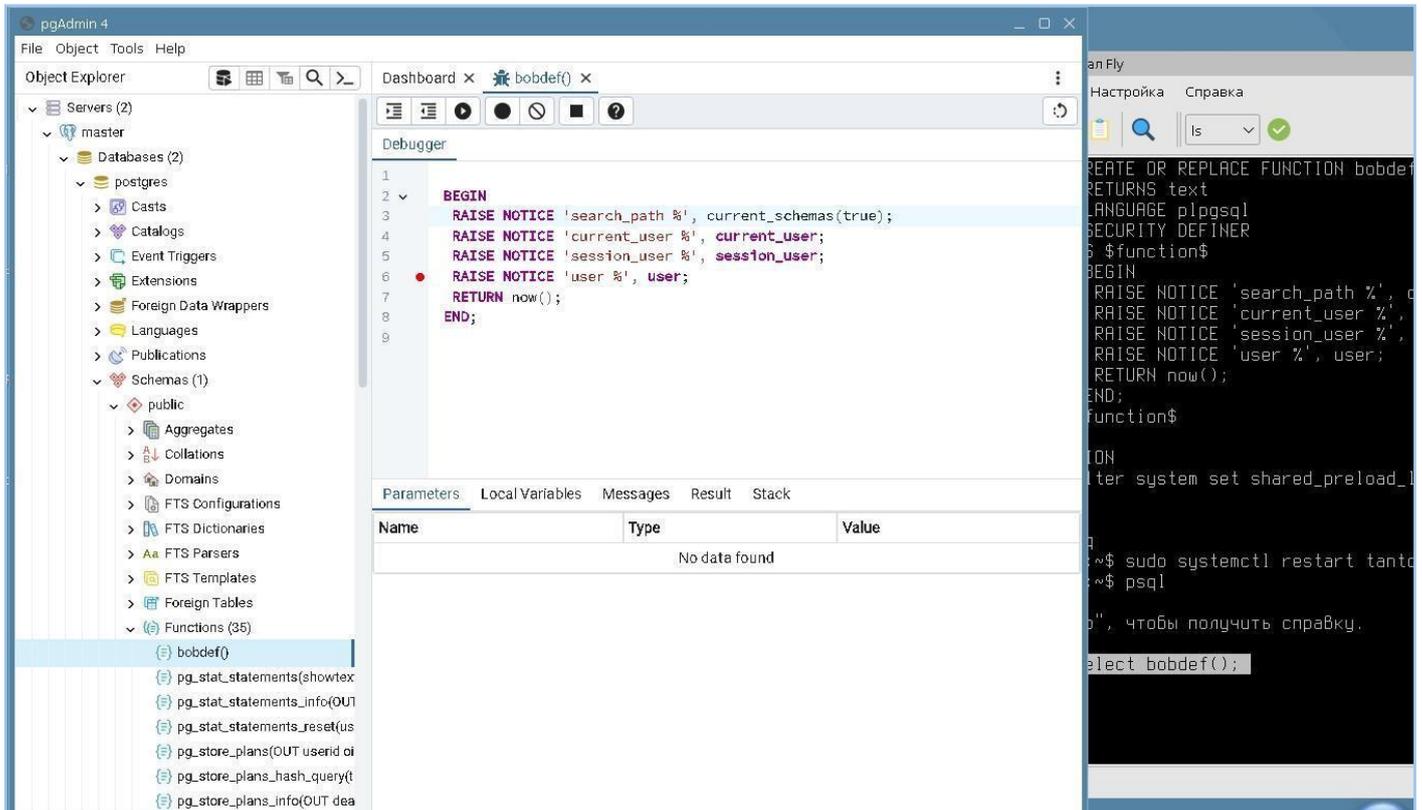
```
postgres=# select bobdef();
```



5) Окно pgAdmin отвиснет и покажет исходный код подпрограммы. Точка останова - первая команда подпрограммы.

В окне с текстом функции можно нажать на иконку (вторую слева) Step over - будет пошаговое выполнение. При этом в окне psql можно наблюдать вывод команд RAISE NOTICE.

Также можно устанавливать точки останова. Для их установки или удаления нужно кликнуть мышкой правее от номера строки. Справа от числа 6 на картинке виден красный кружок - в это место можно кликнуть, и кружок обозначает точку останова.



6) Нажимая мышкой на иконку Step Over или Continue/Start, дойдите до окончания выполнения функции.

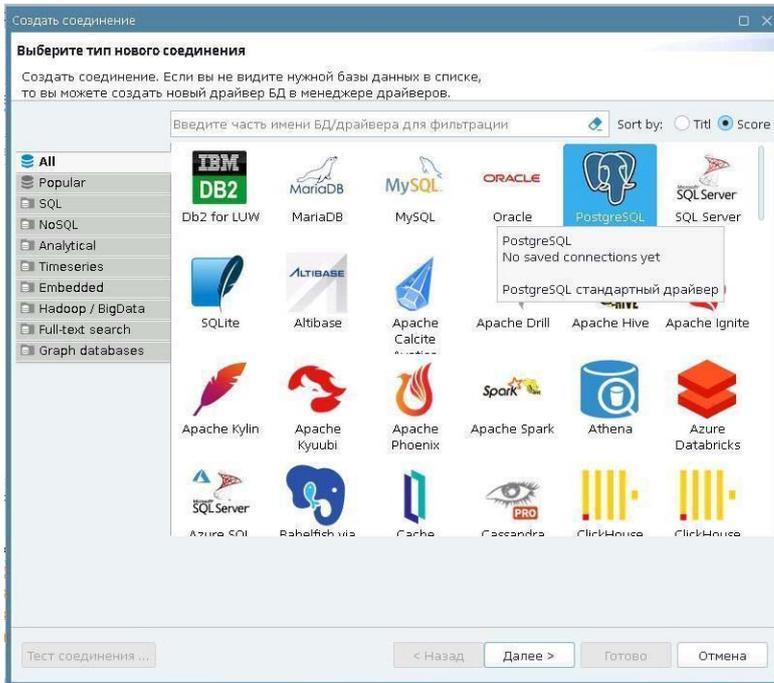
7) Для выполнения отладки с вызовом подпрограммы в сессии pgAdmin, можно выбрать в выпадающем меню Debugging -> Debug. В этом случае не потребуется запускать функцию в psql, она запустится в pgAdmin и в окне pgAdmin будут выдаваться client_messages (результат команд RAISE NOTICE).

Часть 3. Отладка подпрограмм в DBeaver

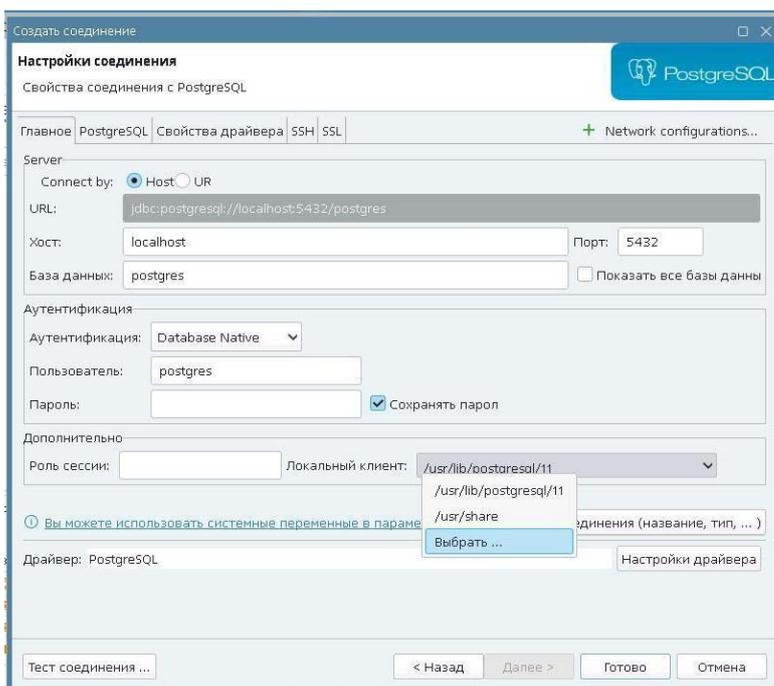
1) Запустите DBeaver. При первом запуске программа предложит создать Sample Database, её создавать не нужно, она не относится к postgres.



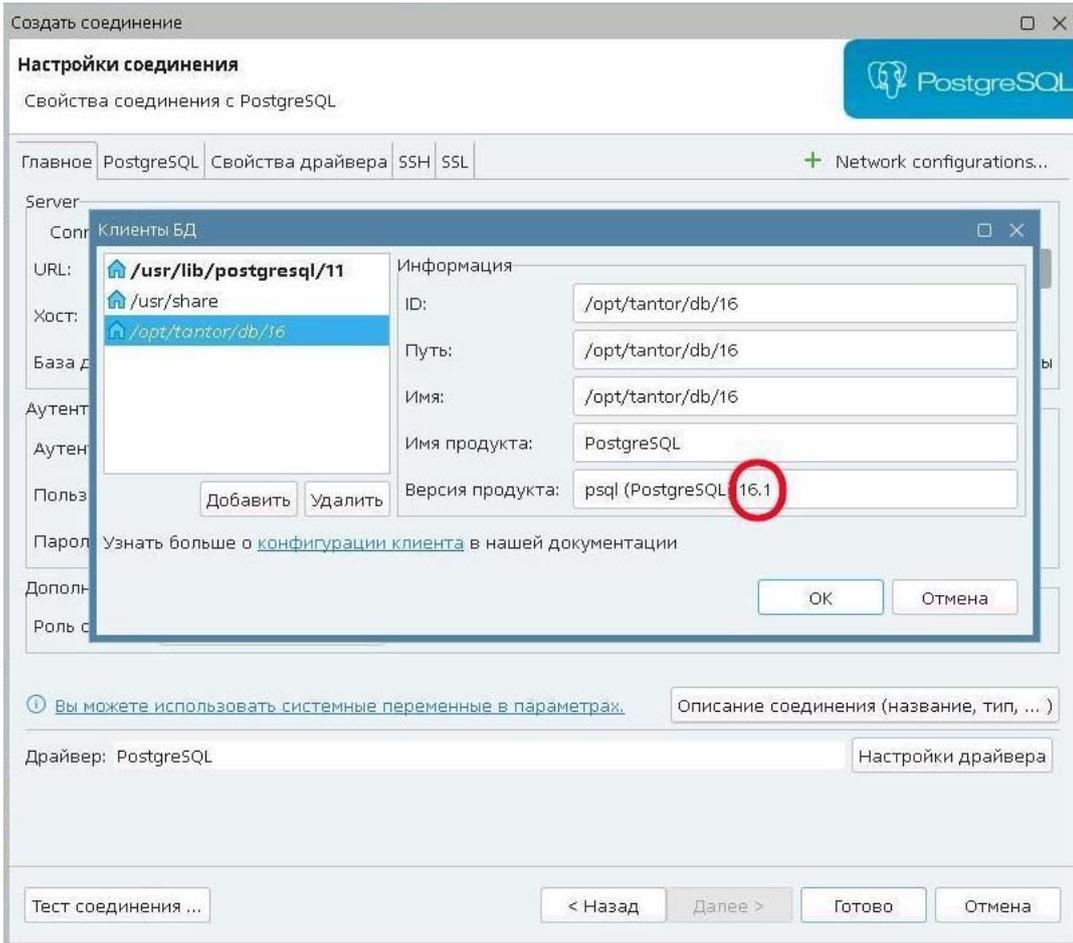
2) Выберите иконку PostgreSQL:



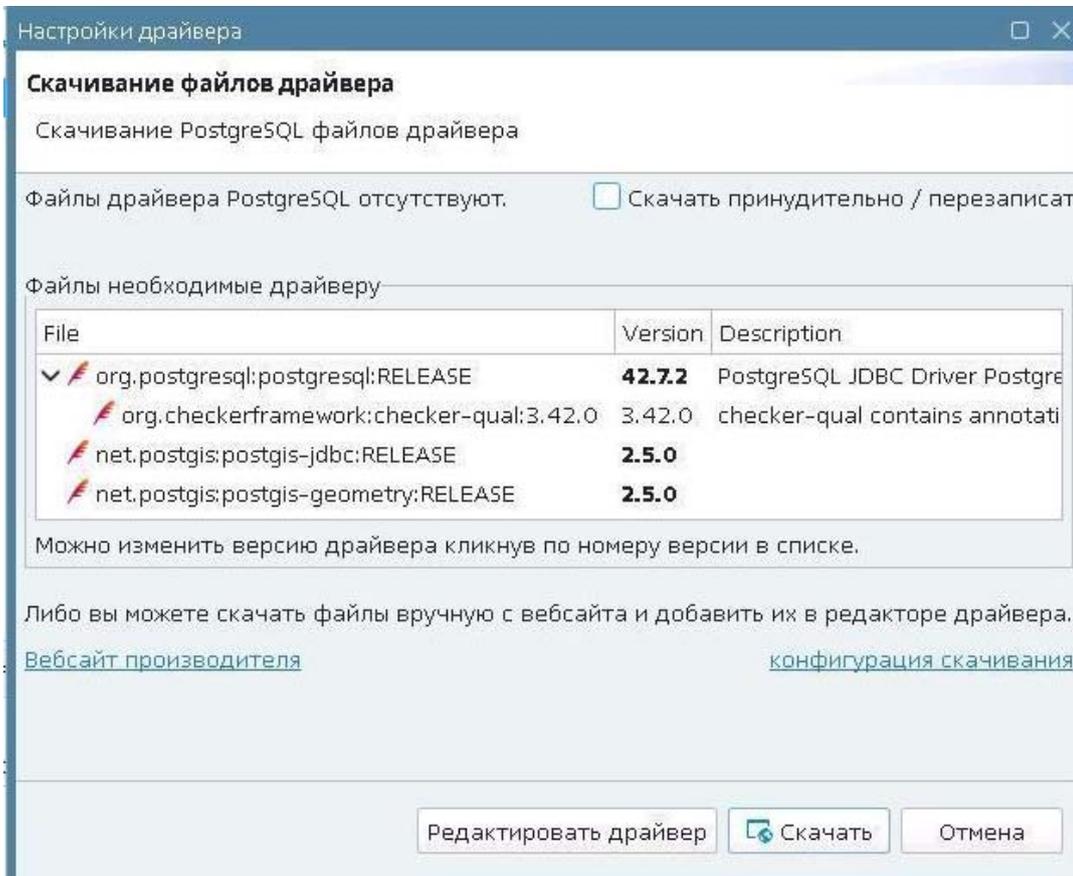
3) Создайте, если не создано ранее, соединение с базой данных. Обратите внимание, что лучше выбрать директорию локального клиента Tantor 16, создав определение клиента по пути `/opt/tantor/db/16:`



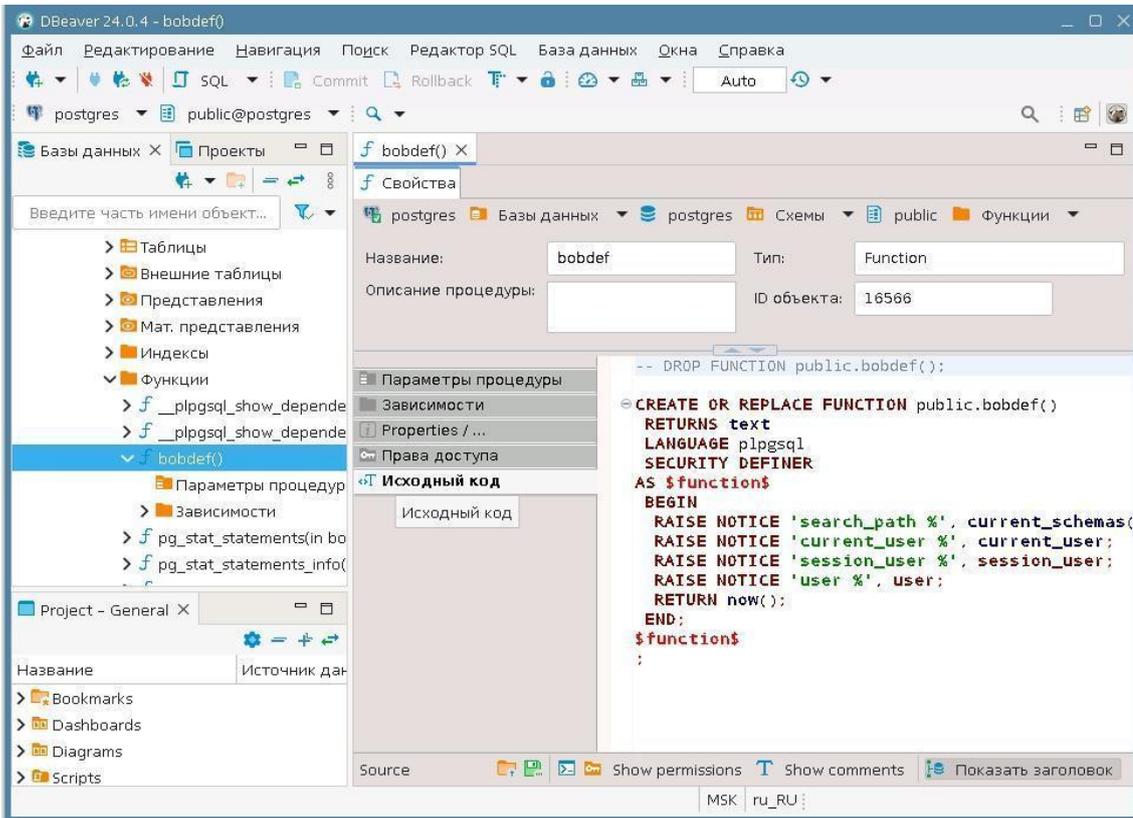
4) Убедитесь, что в поле версии продукта указано число:



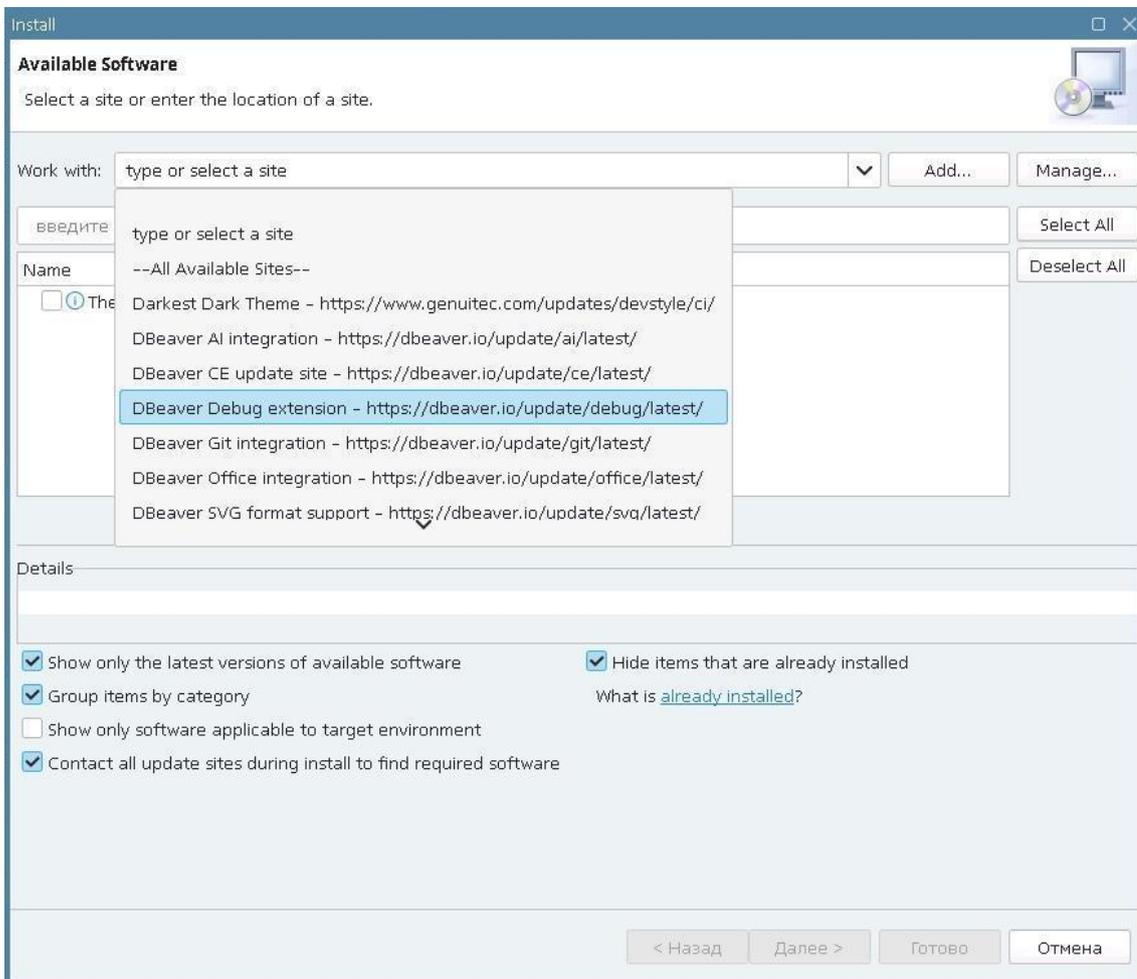
5) DBeaver написан на java и предложит скачать jdbc драйвер:



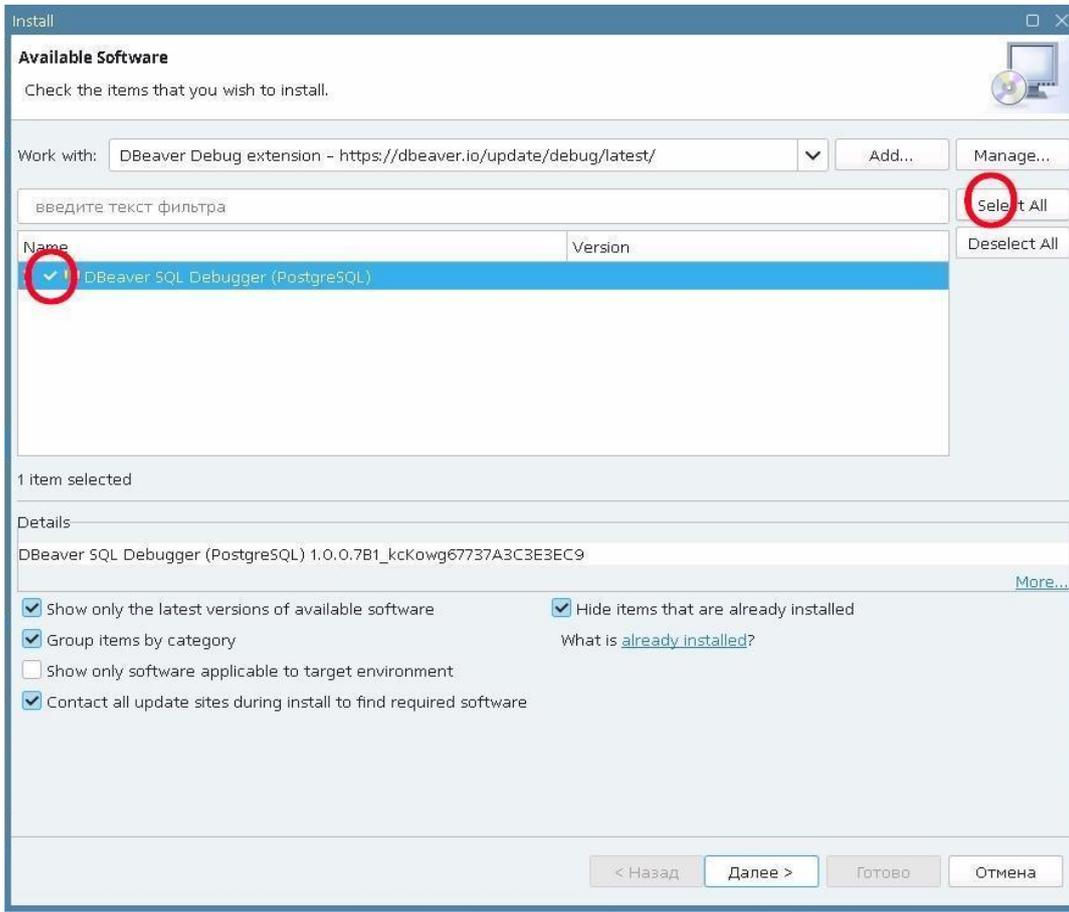
6) Выберите подпрограмму для отладки и кликните на окно "Исходный код":



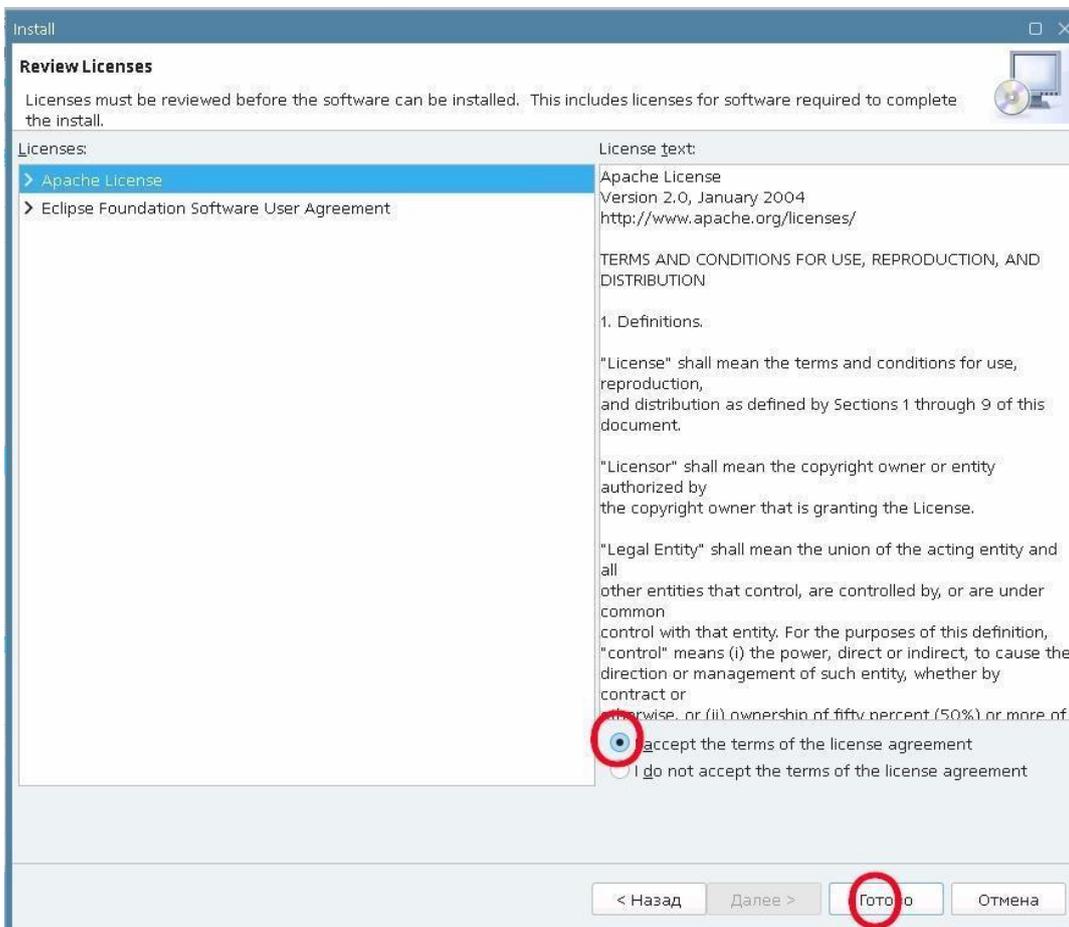
7) Выберите в меню Справка -> Install New Software. Выберите для установки DBeaver Debug Extension:



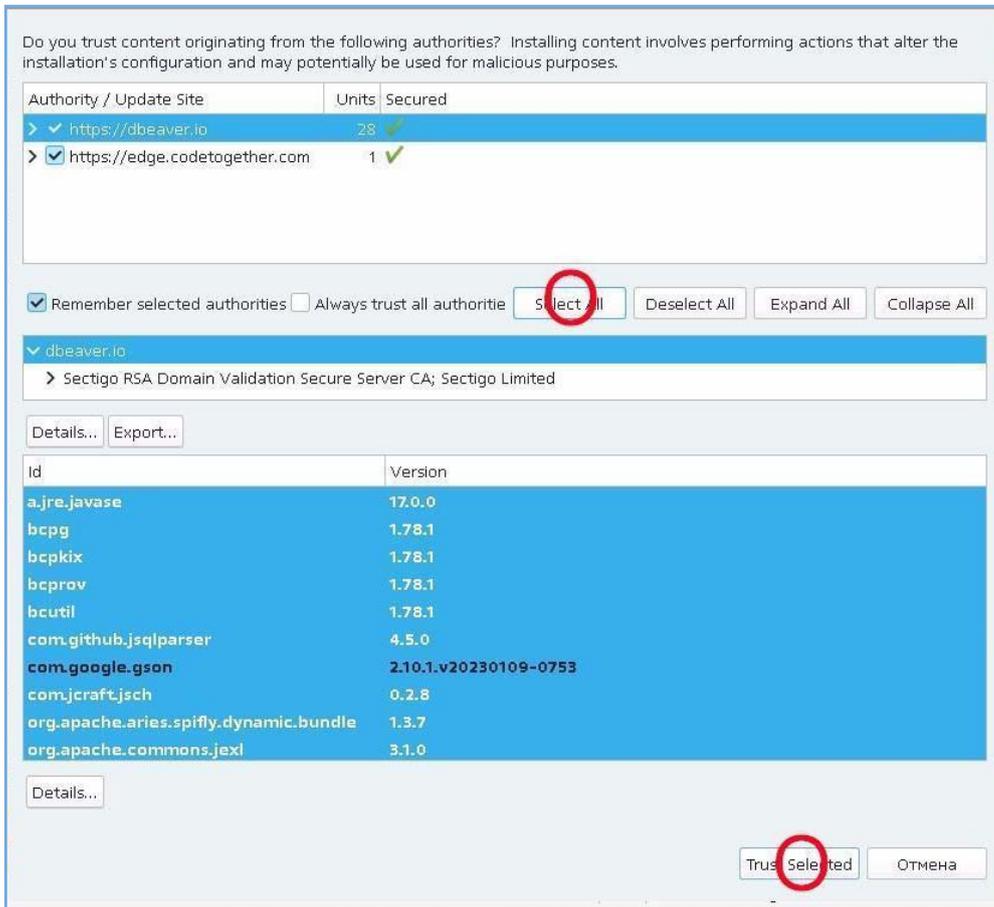
8) Выберите Select All, установится галочка:



9) Выберите точку на radio button:

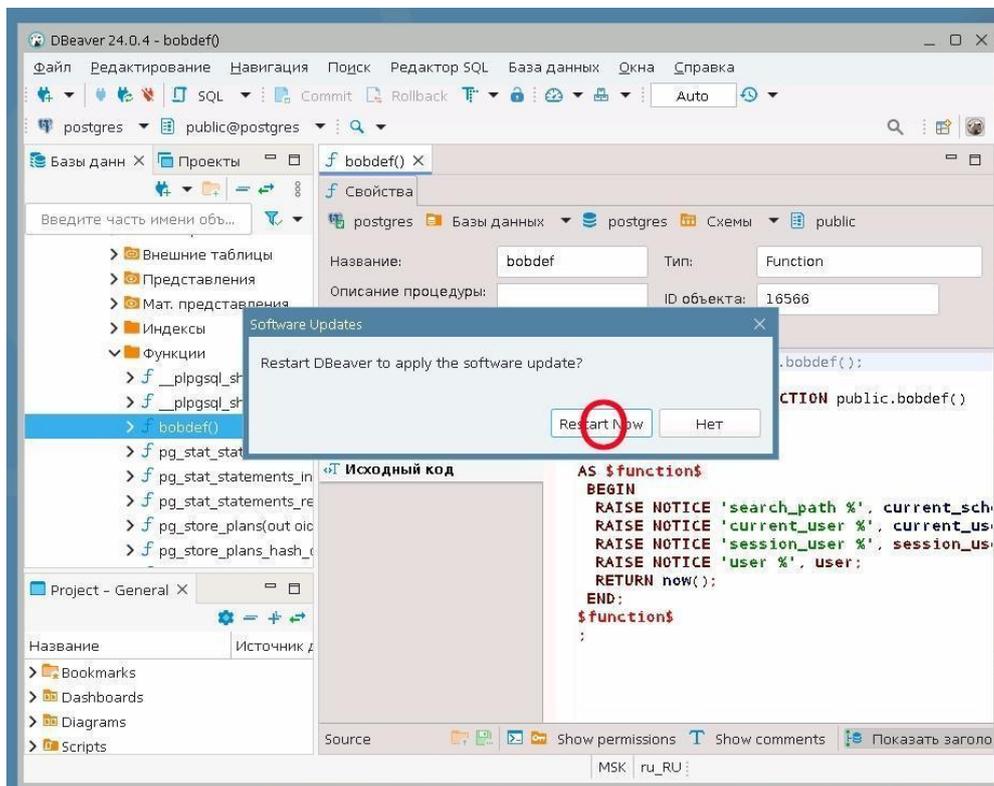


10) В появившемся окне нажмите Select All и кнопку Trust Selected:



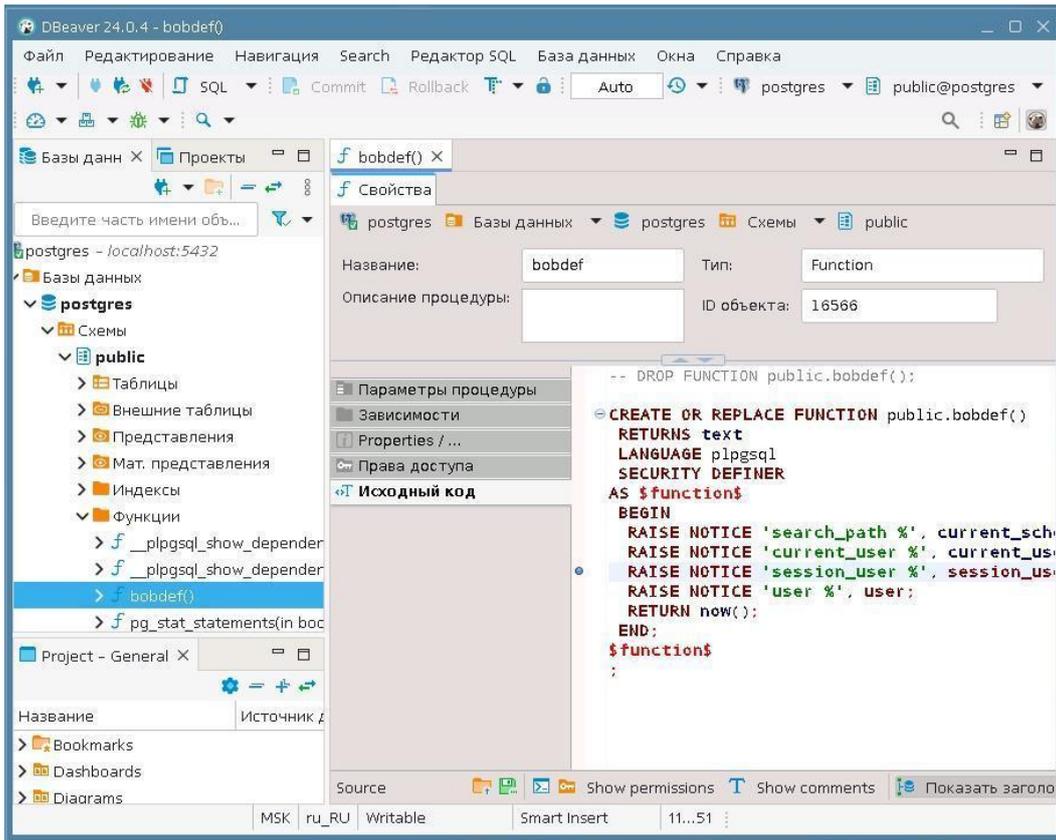
Если утилита повисла и кнопка не нажимается, убейте процесс. Утилита может подвиснуть, если кликнуть в этом окне куда-либо, кроме Select All, и затем Trust Selected.

11) Утилита предложит перезапуститься, перезапустите её:

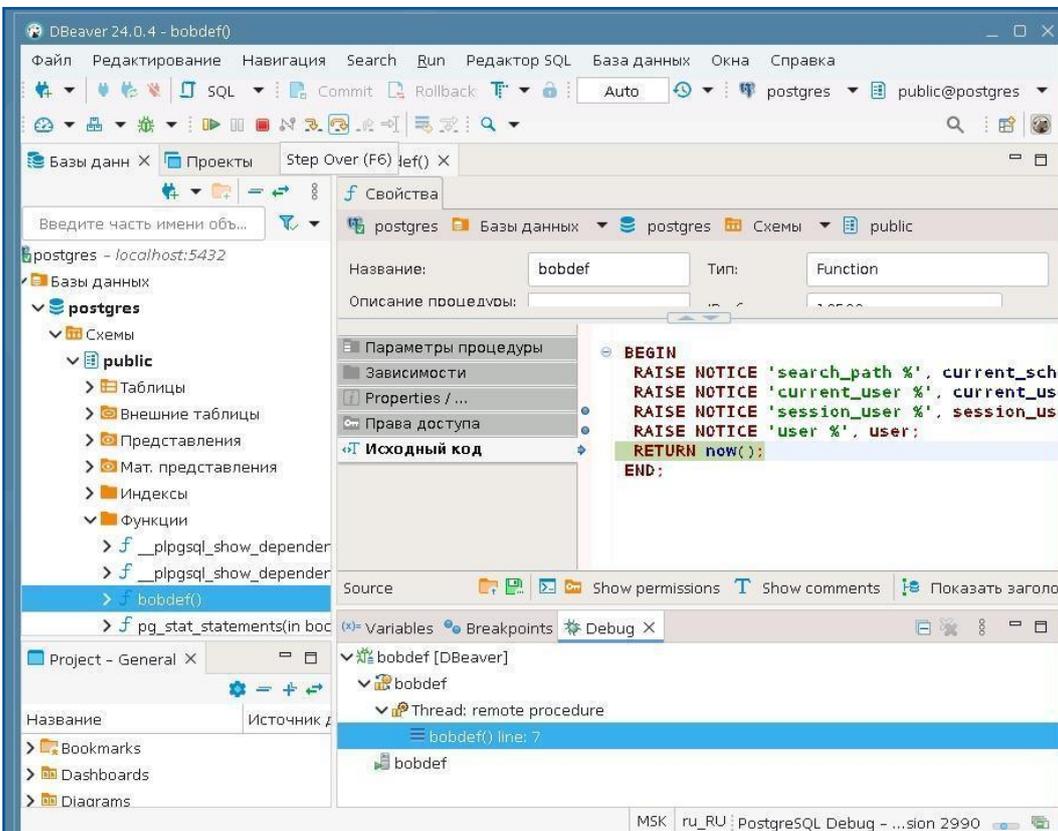


12) Проверьте, что после перезапуска, открыто окно с исходным кодом подпрограммы.

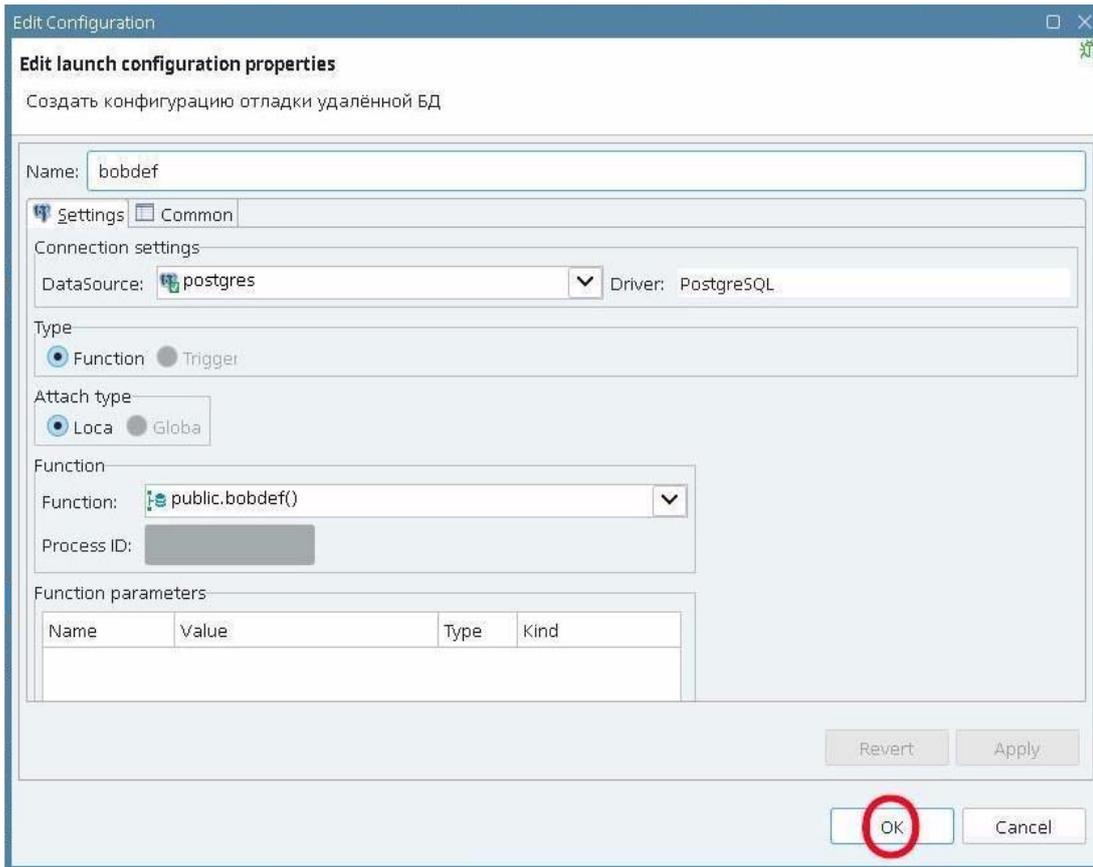
Если не открыто, то выберите подпрограмму и нажмите на вкладку "Исходный код":



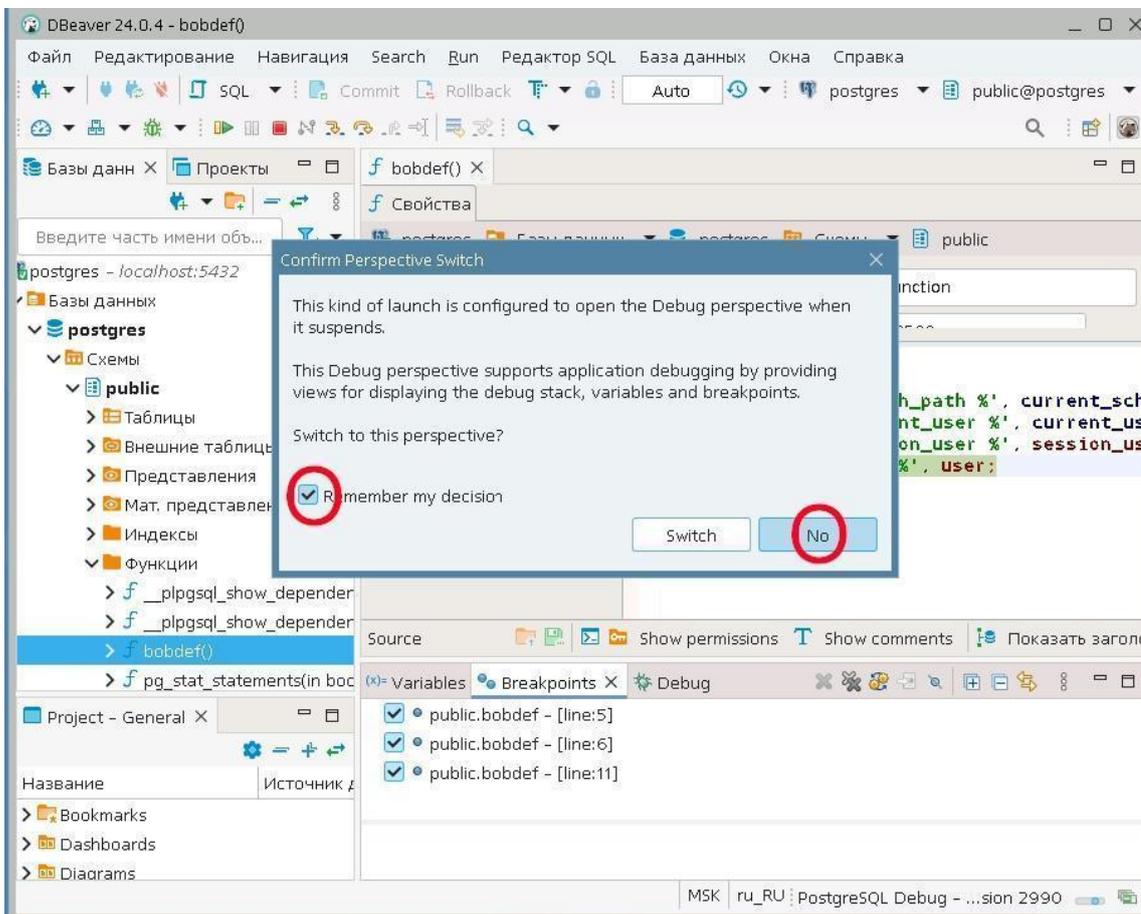
13) Нажмите зелёную иконку на toolbar:



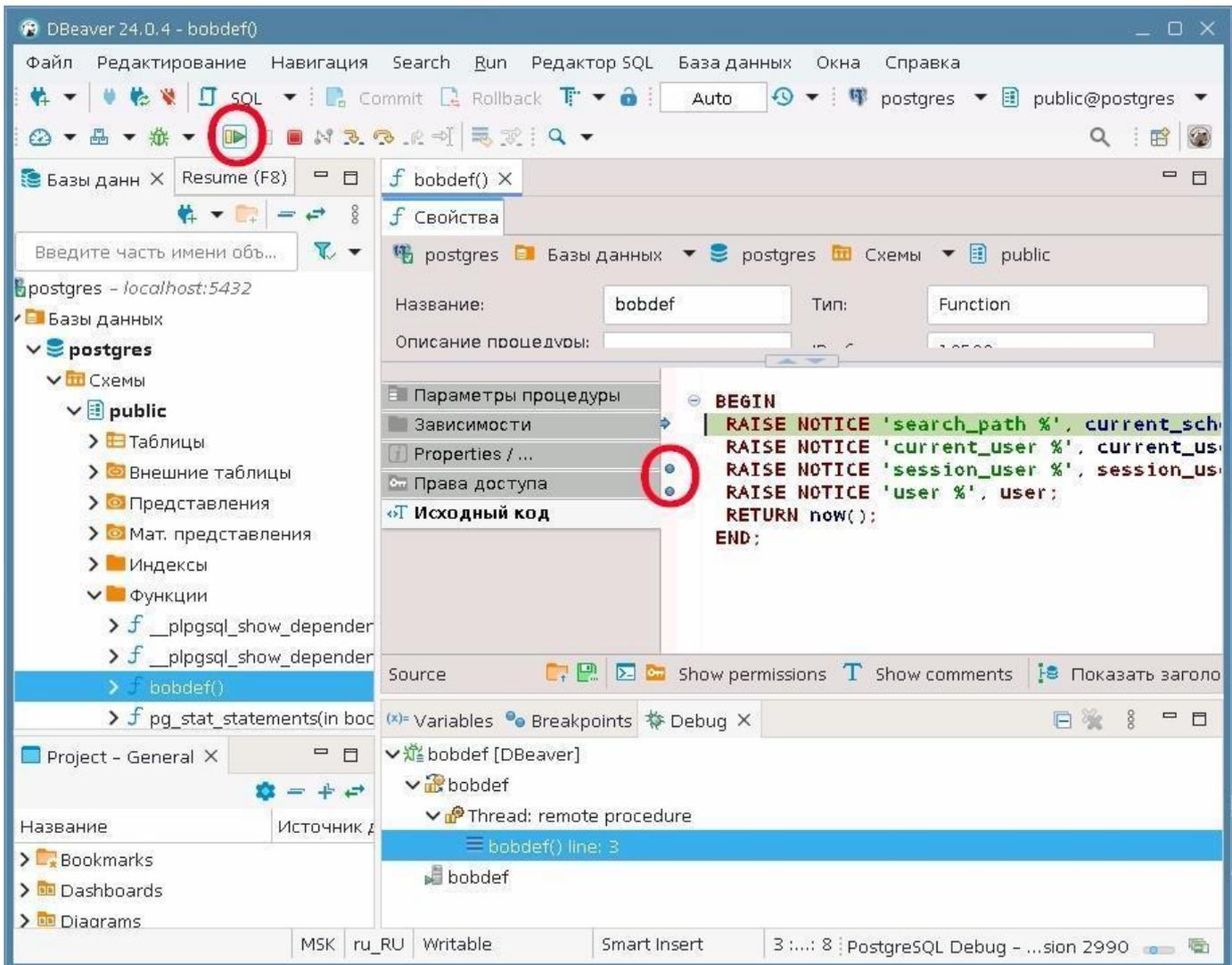
14) В появившемся окне настроек отладки можно установить параметры вызова подпрограммы. Нажмите "OK":



15) В появившемся окне выбора интерфейса отладчика нажмите No:



16) Кликая в начале строк исполняемого кода, можно устанавливать и снимать точки останова, они отображаются синими кружками. С точками останова можно продолжать выполнение кода, нажимая на иконку с зелёным треугольником:



Все иконки на toolbar стандартны для отладчиков в графических средах разработки (IDE): Step into (F5), Step over (F6), Terminate (Ctrl+F2), Resume (F8).

Обработка строк большого размера - StringBuffer

1) Выполните команды:

```
drop table if exists t2;
create table t2 (c1 text, c2 text);
insert into t2 (c1)
VALUES (repeat('a', 1024*1024*512));
update t2 set c2 = c1;
select * from t2;
```

При выполнении команды `select` появится **ошибка**:

```
ERROR: out of memory
```

```
ПОДРОБНОСТИ: Cannot enlarge string buffer containing 536870922 bytes by
536870912 more bytes.
```

При выборке в строковый буфер выбиралось значение поля `c1`, плюс 10 байт. Для выборки значения второго поля `c2` буфер пытался увеличиться на размер поля `c2`.

2) Попробуем с меньшими полями:

```
drop table if exists t1;
create table t1 (c1 text, c2 text, c3 text, c4 text);
insert into t1 (c1) VALUES (repeat('a', 1024*1024*256));
update t1 SET c2=c1;
update t1 SET c3=c1;
update t1 SET c4=c1;
select * from t1;
```

Появится **ошибка**:

```
ERROR: out of memory
```

```
ПОДРОБНОСТИ: Cannot enlarge string buffer containing 805306386 bytes by
268435456 more bytes.
```

При выборке в строковый буфер выбирались значения полей `c1`, `c2`, `c3`. Буфер достиг размера трёх полей плюс 18 байт. При увеличении размера буфера на размер поля `c4`, возникла ошибка превышения границы 1 Гб.

3) Выполните команду:

```
postgres=# COPY t2 TO '/tmp/test';
```

```
ERROR: out of memory
```

```
ПОДРОБНОСТИ: Cannot enlarge string buffer containing 536870913 bytes by
536870912 more bytes.
```

Возникла та же самая ошибка.

4) Строки размером больше 1 Гб можно выгрузить по отдельным столбцам. Тип данных `text` и другие типы данных имеют ограничение на размер поля равное 1 Гб. Выполните команду, выгружающую содержимое одного столбца:

```
postgres=# COPY t2 (c1) TO '/tmp/test';
COPY 1
postgres=# \! ls -al /tmp/test
-rw-r--r-- 1 postgres postgres 536870913 /tmp/test
postgres=# \! rm /tmp/test
```

Содержимое столбца было успешно выгружено.

5) Выполните:

```
drop table if exists t2;
create table t2 (c1 text);
insert into t2 (c1) VALUES (repeat('a\n', 357913941));
COPY t2 TO '/tmp/test';
```

Появится ошибка:

```
postgres=# COPY t2 TO '/tmp/test';
ERROR:  out of memory
ПОДРОБНОСТИ:  Cannot enlarge string buffer containing 1073741822 bytes by 1 more bytes.
```

Ограничение на память строкового буфера было превышено на 1 байт.

Размер поля - треть гигабайта с округлением в меньшую сторону.

При выгрузке в текстовом виде содержимое поля будет выглядеть так:

a\na\na\na\n и размер поля увеличится в три раза до 1073741823 байт, что на 1 байт превышает максимальную границу.

6) При использовании формата binary поле можно выгрузить:

```
postgres=# COPY t2 TO '/tmp/test' WITH BINARY;
COPY 1
postgres=# \! ls -al /tmp/test
-rw-r--r-- 1 postgres postgres 715827909 /tmp/test
postgres=# \! rm /tmp/test
```

7) Посмотрите сколько памяти выделяет серверный процесс при обработке строки.

Выполните команды:

```
drop table if exists t2;
create table t2 (c1 text, c2 text);
insert into t2 (c1) values (repeat('a', 1024*1024*1024-69));
```

В процессе выполнения команды insert, если успеть, то можно во втором окне терминала посмотреть, как менялся объем занятой и свободной памяти (нажимая на клавиатуре клавишу <стрелка вверх> и <Enter>):

```
postgres@tantor:~$ free -b -w
              total          used         free       shared    buffers         cache   available
Mem:      4109729792    633286656  2788950016   148430848   80027648   607465472  3033432064
Swap:              0              0              0
postgres@tantor:~$ free -b -w
              total          used         free       shared    buffers         cache   available
Mem:      4109729792   1280106496  2164342784   148439040   80093184   585187328  2386747392
Swap:              0              0              0
postgres@tantor:~$ free -b -w
              total          used         free       shared    buffers         cache   available
Mem:      4109729792   1514721280  1929728000   148439040   80093184   585187328  2152132608
Swap:              0              0              0
postgres@tantor:~$ free -b -w
              total          used         free       shared    buffers         cache   available
Mem:      4109729792   1948651520  1495797760   148439040   80093184   585187328  1718202368
Swap:              0              0              0
postgres@tantor:~$ free -b -w
              total          used         free       shared    buffers         cache   available
Mem:      4109729792   2772905984   671543296   148439040   80093184   585187328   893947904
Swap:              0              0              0
postgres@tantor:~$ free -b -w
              total          used         free       shared    buffers         cache   available
Mem:      4109729792   656199680   2735239168   148439040   80093184   638197760  3010174976
Swap:              0              0              0
```

Память выделяется динамически.

Использование памяти **увеличилось на ~2 ГБ (2125635584 байт)**. Свободной памяти осталось **~670 МБ**.

8) Если на хосте (виртуальной машине) не хватает физической памяти для выделения буфера обработки строк, то экземпляр может аварийно остановиться. Выполните команды:

```
update t2 set c2 = c1;
select * from t2;
```

сервер неожиданно закрыл соединение

Скорее всего сервер прекратил работу из-за сбоя до или в процессе выполнения запроса.

Подключение к серверу потеряно. Попытка восстановления неудачна.

Подключение к серверу потеряно. Попытка восстановления неудачна.

!> \q

```
postgres@tantor:~$ psql
```

```
psql (17.5)
```

Введите "help", чтобы получить справку.

Такая ошибка возникнет при нехватке физической памяти. Серверный процесс пытался выделить ~4 ГБ памяти, а свободной памяти было меньше 2.7 ГБ. **oom-kill** (out of memory killer) убил серверный процесс. Однако oom-kill может убить произвольные процессы. Процесс postgres остановил все процессы и запустил фоновые процессы.

В процессе динамического выделения памяти операционная система уменьшала размер кэша операционной системы. Если в кэше операционной системы было бы много страниц, не записанных на диск, операционная система пыталась бы их записать и стала бы менее "отзывчивой".

```
postgres@tantor:~$ free -b -w
```

	total	used	free	shared	buffers	cache	available
Mem:	4109729792	3190587392	145354752	148439040	80482304	693305344	474697728
Swap:	0	0	0				

```
postgres@tantor:~$ free -b -w
```

	total	used	free	shared	buffers	cache	available
Mem:	4109729792	3805593600	117968896	148439040	237568	185929728	21350400
Swap:	0	0	0				

```
postgres@tantor:~$ free -b -w
```

	total	used	free	shared	buffers	cache	available
Mem:	4109729792	629743616	3223060480	134189056	4390912	252534784	3134205952
Swap:	0	0	0				

Сообщения в журнале операционной системы:

```
postgres@tantor:~$ sudo dmesg
```

```
[79734.048885] oom-kill:
constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=/,mems_allowed=0-1,global_oom,
task_memcg=/system.slice/tantor-se-server-
17.service,task=postgres,pid=5041,uid=999
[79734.048904] Out of memory: Killed process 5041 (postgres) total-vm:4425648kB,
anon-rss:3177400kB, file-rss:4kB, shmem-rss:34624kB, UID:999 pgtables:6444kB
oom_score_adj:0
```

Сообщения в логе кластера:

```
postgres@tantor:~$ cat $PGDATA/current_logfiles
```

```
stderr log/postgresql-000000.log
```

```
postgres@tantor:~$ tail -n 15 $PGDATA/log/postgresql-000000.log
```

```
[31030] LOG: server process (PID 31038) was terminated by signal 9: Killed
[31030] DETAIL: Failed process was running: select * from t2;
[31030] LOG: terminating any other active server processes
[31030] LOG: all server processes terminated; reinitializing
[31039] LOG: database system was interrupted; last known up at 19:58:59 MSK
[31042] FATAL: the database system is in recovery mode
Failed.
[31039] LOG: database system was not properly shut down; automatic recovery in progress
[31039] LOG: redo starts at 116/CE344C0
[31039] LOG: invalid record length at 116/DF34798: expected at least 26, got 0
[31039] LOG: redo done at 116/DF34770 system usage: CPU: user: 0.02 s, system: 0.12 s, elapsed:
0.15 s
[31040] LOG: checkpoint starting: end-of-recovery immediate wait
[31040] LOG: checkpoint complete: wrote 2105 buffers (12.8%); 0 WAL file(s) added, 0 removed, 0
```

```
recycled; write=0.025 s, sync=0.003 s, total=0.031 s; sync files=25, longest=0.001 s, average=0.001 s; distance=17408 kB, estimate=17408 kB; lsn=116/DF34798, redo lsn=116/DF34798  
[31030] LOG: database system is ready to accept connections
```

oom-kill послал **сигнал 9 (SIGKILL)** серверному процессу, который при выполнении команды `select * from t2` пытался выделить много памяти, но **oom-kill** может послать **сигнал 9 (SIGKILL)** и другим процессам.

Процесс postgres **останавливает все процессы и снова запускает процессы**, как при запуске экземпляра.

9) Удалите таблицы:

```
postgres=# drop table t1;  
DROP TABLE  
postgres=# drop table t2;  
DROP TABLE
```

Поиск осиротевших файлов

В директориях `PGDATA` и табличных пространствах могут присутствовать файлы, которые не используются кластером. Такие файлы могут появляться в результате непредвиденной остановки процесса, создавшего файл. Например, при создании таблицы создаются строки в таблицах системного каталога и файлы в файловой системе. Если процесс сбивнёт, то при рестарте экземпляра строк в таблицах системного каталога не будет, если транзакция не успела зафиксироваться. При этом файлы в файловой системе обычно остаются. Экземпляры postgres не часто некорректно (сигналы `SECKILL`, `SIGSEGV`) останавливаются, поэтому проблема не сильно актуальна с точки зрения места, занимаемого "осиротевшими" в результате исчезновения породившего их процесса файлами. Например, при нехватке памяти `oom-kill` посылает сигнал `SECKILL`. Установим расширение, с помощью которого проверим, есть ли такие файлы в кластере.

1) Выполните команды по установке расширения:

```

astra@tantor:~/pg_orphaned-master$ su - root
Password: root
root@tantor:~# wget
https://github.com/bdrouvot/pg_orphaned/archive/refs/heads/master.zip

HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/bdrouvot/pg_orphaned/zip/refs/heads/master
[following]
https://codeload.github.com/bdrouvot/pg_orphaned/zip/refs/heads/master
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: 'master.zip'
master.zip      [ <=> ]  13.79K  --.-KB/s    in 0.04s
(308 KB/s) - 'master.zip' saved [14119]

root@tantor:~# unzip master.zip
Archive:  master.zip
5038f7ed2579cfbdcelccb4fbac311267b66779a
  creating: pg_orphaned-master/
  inflating: pg_orphaned-master/LICENSE
  inflating: pg_orphaned-master/Makefile
  inflating: pg_orphaned-master/README.md
  inflating: pg_orphaned-master/pg_orphaned--1.0.sql
  inflating: pg_orphaned-master/pg_orphaned.c
  inflating: pg_orphaned-master/pg_orphaned.control

root@tantor:~# cd pg_orphaned-master
root@tantor:~/pg_orphaned-master# export PATH=/opt/tantor/db/17/bin:$PATH
root@tantor:~/pg_orphaned-master# export USE_PGXS=1
root@tantor:~/pg_orphaned-master# make
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -
Werror=vla -Wendif-labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -
Wcast-function-type -Wshadow=compatible-local -Wformat-security -fno-strict-
aliasing -fwrapv -fexcess-precision=standard -Wno-format-truncation -Wno-
stringop-truncation -O2 -pipe -Wno-missing-braces -fPIC -fvisibility=hidden -I. -
I./ -I/opt/tantor/db/17/include/postgresql/server -
I/opt/tantor/db/17/include/postgresql/internal -D_GNU_SOURCE -
I/usr/include/libxml2 -c -o pg_orphaned.o pg_orphaned.c
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -
Werror=vla -Wendif-labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -
    
```

```
Wcast-function-type -Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard -Wno-format-truncation -Wno-stringop-truncation -O2 -pipe -Wno-missing-braces -fPIC -fvisibility=hidden -shared -o pg_orphaned.so pg_orphaned.o -L/opt/tantor/db/17/lib -L/usr/lib/llvm-11/lib -Wl,--as-needed -Wl,-rpath,'/opt/tantor/db/17/lib',--enable-new-dtags -lm -fvisibility=hidden /usr/bin/clang-11 -Wno-ignored-attributes -fno-strict-aliasing -fwrapv -Wno-unused-command-line-argument -O2 -I. -I./ -I/opt/tantor/db/17/include/postgresql/server -I/opt/tantor/db/17/include/postgresql/internal -D_GNU_SOURCE -I/usr/include/libxml2 -flto=thin -emit-llvm -c -o pg_orphaned.bc pg_orphaned.c
```

```
root@tantor:~/pg_orphaned-master# make install
/usr/bin/mkdir -p '/opt/tantor/db/17/lib/postgresql'
/usr/bin/mkdir -p '/opt/tantor/db/17/share/postgresql/extension'
/usr/bin/mkdir -p '/opt/tantor/db/17/share/postgresql/extension'
/usr/bin/install -c -m 755 pg_orphaned.so
'/opt/tantor/db/17/lib/postgresql/pg_orphaned.so'
/usr/bin/install -c -m 644 ./pg_orphaned.control
'/opt/tantor/db/17/share/postgresql/extension/'
/usr/bin/install -c -m 644 ./pg_orphaned--1.0.sql
'/opt/tantor/db/17/share/postgresql/extension/'
/usr/bin/mkdir -p '/opt/tantor/db/17/lib/postgresql/bitcode/pg_orphaned'
/usr/bin/mkdir -p '/opt/tantor/db/17/lib/postgresql/bitcode'/pg_orphaned/
/usr/bin/install -c -m 644 pg_orphaned.bc
'/opt/tantor/db/17/lib/postgresql/bitcode'/pg_orphaned./
cd '/opt/tantor/db/17/lib/postgresql/bitcode' && /usr/lib/llvm-11/bin/llvm-lto -thinlto -thinlto-action=thinlink -o pg_orphaned.index.bc
pg_orphaned/pg_orphaned.bc
```

```
root@tantor:/pg_orphaned-master# exit
logout
astra@tantor:~$ psql
psql (17.5)
Введите "help", чтобы получить справку.
postgres=# CREATE EXTENSION pg_orphaned;
CREATE EXTENSION
```

2) Посмотрите на список функций, которые создало расширение:

```
postgres=# \df *orphane*
Schema | Name | Result data type |
-----+-----+-----+-----
public | pg_list_orphaned | SETOF record | older_than interval DEFAULT
public | pg_list_orphaned_moved | SETOF record | OUT dbname text, OUT path t
public | pg_move_back_orphaned | integer |
public | pg_move_orphaned | integer | older_than interval DEFAULT
public | pg_remove_moved_orphaned | void |
(5 rows)
```

3) Проверьте, есть ли в директориях табличных пространств осиротевшие файлы данных:

```
postgres=# select * from pg_list_orphaned('1 second');
dbname | path | name | size | mod_time | relfilenode | reloid | older
-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

4) Получите PID серверного процесса:

```
postgres=# select pg_backend_pid();
pg_backend_pid
```

```
-----
10555
```

(1 row)

5) В окне psql дайте команды:

```
postgres=# drop table if exists t2;
DROP TABLE
postgres=# begin;
BEGIN
postgres=# create table t2 (c1 text, c2 text);
CREATE TABLE
postgres=# insert into t2 (c1) values (repeat('a', 1024*1024*1024-69));
```

5) Запустите второе окно терминала и подготовьте к выполнению команду, и пошлите сигнал 11 серверному процессу:

```
astra@tantor:~$ sudo kill -11 10555
```

6) Процесс остановился, экземпляр перегрузился. Так как сессия простаивала, то утилита psql не получила уведомления о том, что серверного процесса больше нет. Выполните любую команду:

```
postgres=# \d t2
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```

Утилита psql сообщила, что соединение было закрыто.

Было симулировано падение серверного процесса по ошибке сегментации.

Процесс postgres остановил все процессы и перезапустил экземпляр.

В логе кластера появятся сообщения:

```
LOG:  server process (PID 10555) was terminated by signal 11: Segmentation fault
DETAIL:  Failed process was running: insert into t2 (c1) values (repeat('a',
1024*1024*1024-69));
LOG:  terminating any other active server processes
FATAL:  the database system is in recovery mode
LOG:  all server processes terminated; reinitializing
LOG:  database system was interrupted; last known up at
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/1BCC9070
LOG:  invalid record length at 0/1BCC91D0: expected at least 26, got 0
LOG:  redo done at 0/1BCC9138 system usage: CPU: user: 0.00 s, system: 0.00 s,
elapsed: 0.00 s
LOG:  checkpoint starting: end-of-recovery immediate wait
LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0
removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.004 s; sync files=2,
longest=0.001 s, average=0.001 s; distance=0 kB, estimate=0 kB; lsn=0/1BCC91D0,
redo lsn=0/1BCC91D0
LOG:  database system is ready to accept connections
```

Транзакция по созданию таблицы была незафиксирована и откатилась, но команды создания файлов не откатились - работа с файлами в файловой системе нетранзакционна.

7) Перезапустите psql или создайте новое соединение:

```
postgres=# \q
```

```
postgres@tantor:~$ psql
psql (17.5)
Type "help" for help.
```

8) Проверьте, появились ли осиротевшие файлы:

```
postgres=# select * from pg_list_orphaned('1 second');
 dbname | path | name | size | mod_time | relfilenode | reloid | older
-----+-----+-----+-----+-----+-----+-----+-----
 postgres | base/5 | 49307 | 155648 | 09:12:58+03 | 49307 | 0 | t
 postgres | base/5 | 49303 | 8192 | 09:12:53+03 | 49303 | 0 | t
 postgres | base/5 | 49306 | 12615680 | 09:12:58+03 | 49306 | 0 | t
 postgres | base/5 | 49306_fsm | 24576 | 09:12:52+03 | 49306 | 0 | t
(4 rows)
```

Файлы появились и **занимают место** на диске.

Примечание: В случае если серверный процесс исчез бы в процессе выполнения

последней команды INSERT, ошибки были бы такие:

```
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
!>> \q
```

```
postgres@tantor:~$ psql
psql (17.5)
Type "help" for help.
postgres=# select * from pg_list_orphaned('1 second');
 dbname | path | name | size | mod_time | relfilenode | reloid | older
-----+-----+-----+-----+-----+-----+-----+-----
 postgres | base/5 | 41113 | 0 | | 41113 | 0 | t
 postgres | base/5 | 41117 | 8192 | | 41117 | 0 | t
 postgres | base/5 | 41116 | 0 | | 41116 | 0 | t
(3 rows)
```

9) Удалите осиротевшие файлы функциями расширения:

```
postgres=# select * from pg_move_orphaned('1 second');
 pg_move_orphaned
-----
 4
(1 row)
```

```
postgres=# select * from pg_remove_moved_orphaned();
 pg_remove_moved_orphaned
-----
(1 row)
```

```
postgres=# select * from pg_list_orphaned('1 second');
 dbname | path | name | size | mod_time | relfilenode | reloid | older
-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

```
postgres=# drop table if exists t2;
NOTICE: table "t2" does not exist, skipping
```

DROP TABLE

Таблица **отсутствует**, так как **транзакция**, в которой она создавалась, не была зафиксирована. Серверный процесс создал версии строк в страницах таблиц системного каталога. Журнальные записи о создании версий строк могли попасть в WAL-файлы, после чего страницы с этими записями могли сохраниться на диске, а могли не успеть сохраниться. В зависимости от этого, после рестарта экземпляра в страницах могут присутствовать записи о версиях строк, могут не присутствовать. В любом случае эти версии строк относятся к незафиксированной транзакции и не видны в сессиях. Такие версии строк будут очищены стандартным способом: быстрой очисткой или автовакуумом.

Резервирование и восстановление с использованием WAL-G

Часть 1. Конфигурирование сервера хранения бэкапов minio

Протокол S3 (Simple Storage Service) используется компаниями, предоставляющими услуги по хранению данных большого объема. Приложение, обслуживающее хранилище, может быть установлено в сети предприятия. В этой практике используется приложение `minio`.

1) Проверьте, установлено ли приложение `minio`:

```
postgres@tantor:~$ sudo apt-get install minio
Reading package lists... Done
Building dependency tree
Reading state information... Done
minio is already the newest version (20240321231343.0.0).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

Приложение установлено.

2) Посмотрите содержимое конфигурационного файла `minio`:

```
postgres@tantor:~$ cat /etc/default/minio
MINIO_ROOT_USER=minioadmin
MINIO_ROOT_PASSWORD=minioadmin
MINIO_VOLUMES="/var/local/minio/disk1"
MINIO_SERVER_URL=http://localhost:9000
```

После установки приложения этот файл и директория создаются вручную. В нем указывается **директория**, в которой будут создаваться бэкапы, **имя и пароль** привилегированного пользователя, **порт для веб-оболочки** управления приложением.

3) Запустите службу `minio`:

```
postgres@tantor:~$ sudo systemctl enable --now minio
Created symlink /etc/systemd/system/multi-user.target.wants/minio.service → /lib/systemd/system/minio.service.
```

4) Запустите браузер и откройте адрес <http://127.0.0.1:9000>

Введите имя `minioadmin` и пароль `minioadmin`

Нажмите кнопку Login и сохраните пароль в браузере.

5) Создайте корзину (логический контейнер для классификации бэкапов). Нажмите на ссылку Create Bucket или слева в меню выберите Buckets.

Введите в поле Bucket Name название `bucket1`. Нажмите кнопку Create Bucket. Корзина будет создана.

Создавать корзины и давайте им названия так, чтобы было легко идентифицировать кластер баз данных, который будет резервироваться. Для каждого кластера необходимо создать свою корзину. WAL-G поддерживает backup offloading - резервирование с физической реплики. Для физических реплик создавать отдельные корзины не нужно.

6) Этот пункт опционален. Его можно не выполнять, он не требуется для WAL-G. Пункт иллюстрирует конфигурирование клиентской утилиты, которая не используется WAL-G. Пункт может быть интересен тем, что можно получить доступ к тестовому облачному хранилищу `minio`

по адресу <https://play.min.io>.

```

postgres@tantor:~$ cat << EOF > s3.config
access-key = minioadmin
secret-key = minioadmin
s3-host = localhost
s3-port = 9000
s3-bucket = bucket1
s3-secure = off
EOF
postgres@tantor:~$ sudo chmod 755 /usr/local/bin/mccli
postgres@tantor:~$ /usr/local/bin/mccli alias set local http://127.0.0.1:9000
mccli: Configuration written to `/var/lib/postgresql/.mccli/config.json`. Please
update your access credentials.
mccli: Successfully created `/var/lib/postgresql/.mccli/share`.
mccli: Initialized share uploads `/var/lib/postgresql/.mccli/share/uploads.json`
file.
mccli: Initialized share downloads
`/var/lib/postgresql/.mccli/share/downloads.json` file.
Enter Access Key: minioadmin
Enter Secret Key: minioadmin
Added `local` successfully.
postgres@tantor:~$ cat .mccli/config.json
{
    "version": "10",
    "aliases": {
        "gcs": {
            "url": "https://storage.googleapis.com",
            "accessKey": "YOUR-ACCESS-KEY-HERE",
            "secretKey": "YOUR-SECRET-KEY-HERE",
            "api": "S3v2",
            "path": "dns"
        },
        "local": {
            "url": "http://127.0.0.1:9000",
            "accessKey": "minioadmin",
            "secretKey": "minioadmin",
            "api": "s3v4",
            "path": "auto"
        },
        "play": {
            "url": "https://play.min.io",
            "accessKey": "Q3AM3UQ867SPQQA43P2F",
            "secretKey": "zuf+ttfteS1swRu7BJ86wekitnifILbZam1KYY3TG",
            "api": "S3v4",
            "path": "auto"
        },
        "s3": {
            "url": "https://s3.amazonaws.com",
            "accessKey": "YOUR-ACCESS-KEY-HERE",
            "secretKey": "YOUR-SECRET-KEY-HERE",
            "api": "S3v4",
            "path": "dns"
        }
    }
}
    
```

Можно открыть новую вкладку в браузере и зайти на адрес <https://play.min.io>

Ввести из файла имя `Q3AM3UQ867SPQQA43P2F` и пароль

`zuf+ttfteS1swRu7BJ86wekitnifILbZam1KYY3TG`

На этом сайте можно посмотреть пример работы серверной части minio.

Часть 2. Установка WAL-G

1) Откройте новый терминал и выполните команду:

```
astra@tantor:~$ sudo dpkg -i wal-g-tantor-all_2.0.1-1astrap1.7-1_amd64.deb
```

Выбор ранее не выбранного пакета wal-g-tantor-all.

(Чтение базы данных на данный момент установлено 211859 файлов и каталогов.)

Подготовка к распаковке wal-g-tantor-all_2.0.1-1astrap1.7-1_amd64.deb

Распаковывается wal-g-tantor-all (2.0.1-1astrap1.7-1)

Настраивается пакет wal-g-tantor-all (2.0.1-1astrap1.7-1)

Пакет содержит единственный файл /opt/tantor/usr/bin/wal-g

2) Проверьте [версию WAL-G](#):

```
astra@tantor:~$ wal-g --version
```

```
wal-g version v2.0.1      b7d53dd7      2024.01.12_16:25:53      PostgreSQL
```

3) Посмотрите, как называется [файл параметров WAL-G](#) и его местоположение:

```
postgres@tantor:~$ wal-g | grep config
```

```
--config string      config file (default is $HOME/.walg.json)
```

```
--turbo              Ignore all kinds of throttling defined in config
```

По умолчанию местоположение файла параметров `$HOME/.walg.json`.

4) Создайте [файл параметров WAL-G](#) в [домашнем каталоге](#) пользователя операционной системы postgres:

```
postgres@tantor:~$ cat > .walg.json << EOF
```

```
{
  "AWS_ENDPOINT": "http://127.0.0.1:9000",
  "WALG_S3_PREFIX": "s3://bucket1",
  "AWS_ACCESS_KEY_ID": "minioadmin",
  "AWS_SECRET_ACCESS_KEY": "minioadmin",
  "AWS_S3_FORCE_PATH_STYLE": "true",
  "WALG_COMPRESSION_METHOD": "brotli",
  "WALG_DELTA_MAX_STEPS": "5",
  "PGDATA": "/var/lib/postgresql/tantor-se-17/data",
  "PGHOST": "/var/run/postgresql"
}
```

```
EOF
```

5) Проверьте, что утилита командной строки wal-g может подсоединиться к серверу по протоколу s3. Для этого проверьте список бэкапов:

```
postgres@tantor:~$ wal-g backup-list
```

```
INFO: 2035/06/25 15:51:48.707457 No backups found
```

Если корзина не существует, то будет выдана ошибка. Пример ошибки:

```
ERROR: 2035/06/25 15:43:19.396870 failed to list s3 folder: 'basebackups_005/':
```

```
NoSuchBucket: The specified bucket does not exist
```

```
status code: 404, request id: 17DF031B83CC101C, host id:
```

```
dd9025bab4ad464b049177c95eb6ebf374d3b3fd1af9251148b658df7ac2e3e8
```

6) Посмотрите, какие WAL-сегменты есть у кластера:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
```

```
0000000100000000000000019 archive_status
```

7) Посмотрим, какой командой передаются WAL-сегменты.

Выполните команду, передав в качестве параметра [имя файла WAL-сегмента](#):

```
postgres@tantor:~$ wal-g wal-push $PGDATA/pg_wal/0000000100000000000000019
```

```
INFO: 2035/06/25 FILE PATH: 0000000100000000000000019.br
```

8) Утилита не удаляет то, что бэкапит. Проверьте, что исходный файл не был удалён:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
00000001000000000000000019 archive_status walg_data
```

В директории журналов была создана поддиректория `walg_data/walg_archive_status`

9) Выполните команду:

```
postgres@tantor:~$ wal-g backup-list
INFO: 2035/06/25 16:01:30.875297 No backups found
```

Бэкапов кластера не найдено, так как мы их пока не делали.

10) Выполните команды:

```
postgres@tantor:~$ wal-g wal-show
INFO: 2035/06/25 No backups found in storage.
```

TLI	PARENT TLI	SWITCHPOINT LSN	START SEGMENT	END SEGMENT	SEGMENT RANGE	SEGMENTS COUNT	STATUS	BACKUPS COUNT
1	0	0/0	000000010000000000000019	000000010000000000000019	1	1	OK	0

```
postgres@tantor:~$ wal-g wal-verify timeline
WARNING: 2035/06/25 16:03:17.728025 It seems your archive_mode is not enabled. This will
cause inconsistent backup. Please consider configuring WAL archiving.
INFO: 2035/06/25 16:03:17.750944 000000010000000000000019
INFO: 2035/06/25 16:03:17.762117 Building check runner: timeline
INFO: 2035/06/25 16:03:17.762253 Running the check: timeline
[wal-verify] timeline check status: OK
[wal-verify] timeline check details:
Highest timeline found in storage: 1
Current cluster timeline: 1
```

```
postgres@tantor:~$ wal-g wal-verify integrity
WARNING: 2035/06/25 16:03:43.575246 It seems your archive_mode is not enabled. This will
cause inconsistent backup. Please consider configuring WAL archiving.
INFO: 2035/06/25 16:03:43.579362 Current WAL segment: 000000010000000000000019
INFO: 2035/06/25 16:03:43.586333 Building check runner: integrity
WARNING: 2035/06/25 16:03:43.604241 Failed to detect earliest backup WAL segment no: 'No
backups found', will scan until the 0000000X000000000000000001 segment.
INFO: 2035/06/25 16:03:43.604378 Running the check: integrity
[wal-verify] integrity check status: WARNING
[wal-verify] integrity check details:
```

TLI	START	END	SEGMENTS COUNT	STATUS
1	000000010000000000000001	000000010000000000000008	8	MISSING_LOST
1	000000010000000000000009	000000010000000000000018	16	MISSING_UPLOADING

Часть 3. Конфигурирование кластера для архивирования журналов

1) Запустите `psql` и установите значения параметров конфигурации:

```
postgres@tantor:~$ psql
psql (17.5)
Type "help" for help.
postgres=# alter system set archive_command = '/opt/tantor/usr/bin/wal-g wal-push "%p" >>
$PGDATA/log/archive_command.log 2>&1';
ALTER SYSTEM
postgres=# alter system set restore_command = '/opt/tantor/usr/bin/wal-g wal-fetch "%f"
"%p" >> $PGDATA/log/restore_command.log 2>&1';
ALTER SYSTEM
postgres=# alter system set archive_mode=on;
ALTER SYSTEM
```

Параметр `archive_command` устанавливает команду, которая будет выполняться после переключения на следующий WAL-сегмент. Команда должна завершаться успешно (возвращать статус "0"), иначе сегмент будет считаться не заархивированным и не сможет удаляться. `%p` - переменная которая инициализируется названием и путём к WAL-сегменту, в который завершена запись, и который должен быть заархивирован. Сообщения утилиты, которые она выводит в `stdout` и `stderr` направляются в файл.

Параметр `restore_command` указывает, какая команда будет выполнена процессом `startup`, который восстанавливает кластер после запуска экземпляра и определяет по файлу `backup_label` или `pg_control`, какой WAL-сегмент нужен для продолжения восстановления (будет накатываться следующим). Эта команда должна создать WAL-файл в директории `$PGDATA/pg_wal`.

Параметр `archive_mode` включает действие параметра `archive_command`.

У этого параметра есть еще значение `always`, что означает, что `archive_command` будет выполняться и во время восстановления из бэкапа, и в режиме физической реплики.

Утилита `wal-g` использует файл параметров `$HOME/.walg.json`. Если нужно иметь несколько файлов параметров, то можно использовать параметр `--config`. Пример:

```
alter system set archive_command = '/opt/tantor/usr/bin/wal-g --config
/var/lib/postgresql/.walg.json wal-push "%p" >> $PGDATA/log/archive_command.log 2>&1';
```

2) Перезапустите экземпляр:

```
postgres@tantor:~$ pg_ctl stop
postgres@tantor:~$ sudo systemctl start tantor-se-server-17
```

3) Проверьте, что в директории журналов появилась поддиректория `archive_status`:

```
postgres@tantor:~$ ls -a $PGDATA/pg_wal/archive_status
. .. 000000010000000000000001A.done
```

В директории присутствует **пустой файл**.

4) Проверьте, что появился файл `archive_command.log`, путь к которому был указан в параметре `archive_command`:

```
postgres@tantor:~$ cat $PGDATA/log/archive_command.log
INFO: 2035/06/25 16:34:37.373971 FILE PATH: 000000010000000000000001A.br
```

Если файл не появился и опечаток нет, это может означать, что экземпляр не был

остановлен, а был перезапущен опцией `restart` (процесс `postgres` не выгружался из памяти).

5) Выполните команду:

```
postgres@tantor:~$ wal-g wal-verify integrity
INFO: 2035/06/25 16:43:29.235139 Current WAL segment: 000000010000000000000001B
INFO: 2035/06/25 16:43:29.244838 Building check runner: integrity
WARNING: 2035/06/25 16:43:29.266445 Failed to detect earliest backup WAL segment no: 'No
backups found', will scan until the 0000000X000000000000000001 segment.
INFO: 2035/06/25 16:43:29.266730 Running the check: integrity
[wal-verify] integrity check status: WARNING
[wal-verify] integrity check details:
+-----+-----+-----+-----+-----+
| TLI | START | END | SEGMENTS COUNT | STATUS |
+-----+-----+-----+-----+-----+
| 1 | 000000010000000000000001 | 00000001000000000000000A | 10 | MISSING_LOST |
| 1 | 00000001000000000000000B | 000000010000000000000018 | 14 | MISSING_UPLOADING |
| 1 | 0000000100000000000000019 | 00000001000000000000001A | 2 | FOUND |
+-----+-----+-----+-----+-----+
```

Файлы журнала были заархивированы.

6) Зарезервируйте директорию кластера. WAL-G запускается на хосте с кластером, поэтому можно не пользоваться протоколом репликации, а копировать содержимое директории.

Для этого нужно передать в качестве параметра название директории кластера:

```
postgres@tantor:~$ wal-g backup-push $PGDATA
INFO: 2035/06/25 16:45:29.484805 Couldn't find previous backup. Doing full
backup.
INFO: 2035/06/25 16:45:29.514046 Calling pg_start_backup()
INFO: 2035/06/25 16:45:29.630427 Starting a new tar bundle
INFO: 2035/06/25 16:45:29.630574 Walking ...
INFO: 2035/06/25 16:45:29.632062 Starting part 1 ...
INFO: 2035/06/25 16:45:35.436669 Packing ...
INFO: 2035/06/25 16:45:35.439440 Finished writing part 1.
INFO: 2035/06/25 16:45:35.814169 Starting part 2 ...
INFO: 2035/06/25 16:45:35.814216 /global/pg_control
INFO: 2035/06/25 16:45:35.816267 Finished writing part 2.
INFO: 2035/06/25 16:45:35.816305 Calling pg_stop_backup()
INFO: 2035/06/25 16:45:35.864195 Starting part 3 ...
INFO: 2035/06/25 16:45:35.868956 backup_label
INFO: 2035/06/25 16:45:35.869596 tablespace_map
INFO: 2035/06/25 16:45:35.871637 Finished writing part 3.
INFO: 2035/06/25 16:45:35.989039 Wrote backup with name
base_000000010000000000000001C
```

В начале резервирования, при вызове функции `pg_start_backup()` была выполнена контрольная точка в режиме `immediate force wait`

```
postgres@tantor:~/tantor-se-17/data/log$ tail -n 2 postgresql-2024-07-04_163447.log
2024-07-04 16:45:29.581 MSK [2973] LOG: checkpoint starting: immediate force wait
2024-07-04 16:45:29.625 MSK [2973] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0
WAL file(s) added, 0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.045 s;
sync files=0, longest=0.000 s, average=0.000 s; distance=16383 kB, estimate=16383 kB;
lsn=0/1C000070, redo lsn=0/1C000028
```

7) Проверьте, что созданный бэкап есть в списке:

```
postgres@tantor:~$ wal-g backup-list
name modified wal_segment_backup_start
base_000000010000000000000001C 2035-06-25T13:45:35Z 000000010000000000000001C
```

Если при резервировании указать произвольную директорию, а не директорию кластера, выдастся ошибка:

```

postgres@tantor:~$ wal-g backup-push abcd
WARNING: 2035/06/25 16:54:15.130694 Data directory for postgres 'abcd' is not equal to
backup-push argument '/var/lib/postgresql/tantor-se-17/data'
ERROR: 2035/06/25 16:54:15.131420 Data directory read from Postgres (abcd) is different
than as parsed (/var/lib/postgresql/tantor-se-17/data).
panic: Data directory read from Postgres (abcd) is different than as parsed
(/var/lib/postgresql/tantor-se-17/data).
    
```

Интуитивно не понятно, что обозначается "Postgres" и "parsed", вероятно, директории перепутаны местами.

Для просмотра бэкапов через веб-интерфейс, нужно на странице корзины кликнуть на иконку папки, справа, вверху окна браузера:

8) В директории `pg_wal` появился **файл**:

```

postgres@tantor:~$ ls $PGDATA/pg_wal/
000000010000000000000001C.00000028.backup  000000010000000000000001E  archive_status
000000010000000000000001D                000000010000000000000001F  walg_data
    
```

```

postgres@tantor:~$ cat $PGDATA/pg_wal/*.backup
START WAL LOCATION: 0/1C000028 (file 000000010000000000000001C)
STOP WAL LOCATION: 0/1C000130 (file 000000010000000000000001C)
CHECKPOINT LOCATION: 0/1C000070
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2035-06-25 16:45:29 MSK
LABEL: 2035-06-25 16:45:29.514037 +0300 MSK m=+0.118109848
START TIMELINE: 1
STOP TIME: 2035-06-25 16:45:35 MSK
STOP TIMELINE: 1
    
```

Часть 4. Восстановление из бэкапа, созданного WAL-G

В конфигурации есть физическая реплика. Проверьте, что слот репликации используется:

```
postgres=# select * from pg_stat_replication\gx
-[ RECORD 1 ]-----+-----
pid           | 2981
usesysid     | 16384
username     | replicator
application_name | walreceiver
client_addr  | ::1
client_hostname |
client_port  | 41306
backend_start | 2035-06-25 16:34:47.297025+03
backend_xmin  |
state        | streaming
sent_lsn     | 0/1D000198
write_lsn    | 0/1D000198
flush_lsn    | 0/1D000198
replay_lsn   | 0/1D000198
write_lag    |
flush_lag    |
replay_lag   |
sync_priority | 0
sync_state   | async
reply_time   | 2035-06-25 17:04:43.201876+03

postgres=# select * from pg_replication_slots \gx
-[ RECORD 1 ]-----+-----
slot_name      | pgstandby1
plugin         |
slot_type     | physical
datoid         |
database      |
temporary     | f
active        | t
active_pid    | 2981
xmin          |
catalog_xmin  |
restart_lsn   | 0/1D000198
confirmed_flush_lsn |
wal_status    | reserved
safe_wal_size | 1090518632
two_phase     | f
conflicting   |
```

2) Если предыдущая часть этой практики была успешно выполнена, то есть архивирование журналов выполнялось, бэкап был сделан, то остановите кластер и удалите директорию PGDATA:

```
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$ rm -rf $PGDATA/*
```

Команда симулирует полную потерю мастера ("disaster"). Команда удаляет в том числе и текущий WAL-сегмент, который не был записан в архив. Транзакции, которые находятся в этом файле, не будут восстановлены.

2) Выполните команду восстановления директории кластера из бэкапа:

```
postgres@tantor:~$ wal-g backup-fetch $PGDATA LATEST
INFO: 2035/06/25 17:09:21.604287 Selecting the latest backup...
INFO: 2035/06/25 17:09:21.608464 LATEST backup is: 'base_000000010000000000000001c'
INFO: 2035/06/25 17:09:21.648710 Finished extraction of part_003.tar.br
```

```
INFO: 2035/06/25 17:09:28.920711 Finished extraction of part_001.tar.br
INFO: 2035/06/25 17:09:28.926843 Finished extraction of pg_control.tar.br
INFO: 2035/06/25 17:09:28.927310
Backup extraction complete.
```

Директория восстановлена.

3) Проверьте содержимое директории журналов:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
```

Директория пуста.

4) Посмотрите содержимое управляющего файла:

```
postgres@tantor:~$ pg_controldata
pg_control version number:          1300
Catalog version number:            202307071
Database system identifier:         7353194261070147214
Database cluster state:         in production
pg_control last modified:      Thu 25 Jun 35 04:45:29 PM MSK
Latest checkpoint location:         0/1C000070
Latest checkpoint's REDO location:  0/1C000028
Latest checkpoint's REDO WAL file:  0000000100000000000000001C
Latest checkpoint's TimeLineID:     1
Latest checkpoint's PrevTimeLineID: 1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:        764
Latest checkpoint's NextOID:        16529
Latest checkpoint's NextMultiXactId: 1
Latest checkpoint's NextMultiOffset: 0
Latest checkpoint's oldestXID:      723
Latest checkpoint's oldestXID's DB:  1
Latest checkpoint's oldestActiveXID: 764
Latest checkpoint's oldestMultiXid:  1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:          Thu 25 Jun 35 04:45:29 PM MSK
Fake LSN counter for unlogged rels:  0/3E8
Minimum recovery ending location:    0/0
Min recovery ending loc's timeline:  0
Backup start location:               0/0
Backup end location:                 0/0
End-of-backup record required:       no
wal_level setting:                   replica
wal_log_hints setting:               off
max_connections setting:              100
max_worker_processes setting:        8
max_wal_senders setting:              10
max_prepared_xacts setting:          0
max_locks_per_xact setting:           64
track_commit_timestamp setting:      off
Maximum data alignment:               8
Database block size:                 8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
Date/time type storage:                64-bit integers
Float8 argument passing:              by value
Data page checksum version:           0
Mock authentication nonce:
5c6cff5b22f9ce01ca3ad035abc27d15de5499782456494a5dbc7016b5fdc3a9
```

Управляющий файл - образ того, который существовал на момент резервирования.

На это указывают строки:

```
Database cluster state:          in production
pg_control last modified:      Thu 25 Jun 35 04:45:29 PM MSK
```

5) Вместо управляющего файла будет использоваться backup_label.

Посмотрите что он присутствует и не пуст:

```
postgres@tantor:~$ cat $PGDATA/backup_label
START WAL LOCATION: 0/1C000028 (file 000000010000000000000001C)
CHECKPOINT LOCATION: 0/1C000070
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2035-06-25 16:45:29 MSK
LABEL: 2035-06-25 16:45:29.514037 +0300 MSK m=+0.118109848
START TIMELINE: 1
```

6) Попробуйте запустить экземпляр:

```
postgres@tantor:~$ pg_ctl start
waiting for server to start....
2024-07-04 17:16:12.130 MSK [3368] СООБЩЕНИЕ:  передача вывода в протокол процессу сбора
протоколов
2024-07-04 17:16:12.130 MSK [3368] ПОДСКАЗКА:  В дальнейшем протоколы будут выводиться в
каталог "log".
stopped waiting
pg_ctl: could not start server
Examine the log output.
```

Экземпляр не смог запуститься, нет журналов для синхронизации файлов кластера, так как в директории pg_wal нет ни одного файла журнала.

7) Создайте файл, который укажет, что выполняется восстановление из бэкапа:

```
postgres@tantor:~$ touch $PGDATA/recovery.signal
```

8) Запустите экземпляр:

```
postgres@tantor:~$ pg_ctl start
waiting for server to start....
2024-07-04 17:21:25.081 MSK [3445] СООБЩЕНИЕ:  передача вывода в протокол процессу сбора
протоколов
2024-07-04 17:21:25.081 MSK [3445] ПОДСКАЗКА:  В дальнейшем протоколы будут выводиться в
каталог "log".
. done
server started
```

Кластер был восстановлен и экземпляр запущен. **Последний примененный журнальный файл тот, который был заархивирован. Было выполнено неполное восстановление. Журнальные файлы, которые не попали в архив, не были применены, и транзакции, которые могли бы в них быть, были потеряны.**

9) Посмотрите содержимое директории с журналами:

```
postgres@tantor:~$ ls -a $PGDATA/pg_wal
.      000000020000000000000001E  0000000200000000000000020  archive_status  walg_data
..     000000020000000000000001F  00000002.history           .wal-g
```

Директория непустая. Была создана **новая** линия времени.

Помимо прочего, появились файл `0000002.history` и пустая директория `.wal-g/prefetch/running`

10) Проверьте, что слоты репликации, которые существовали в начале этой части практики, были удалены:

```
postgres@tantor:~$ psql
```

```
psql (17.5)  
Type "help" for help.
```

```
postgres=# select * from pg_stat_replication\gx  
(0 rows)
```

```
postgres=# select * from pg_replication_slots \gx  
(0 rows)
```

Причина: неполное восстановление.

Часть 5. Использование WAL-G с файловой системой

WAL-G, кроме использования протокола S3, может резервировать и восстанавливать из директории в файловой системе. Директория не обязательно находится на локальном диске, можно смонтировать любую файловую систему, например, NFS.

1) Создайте файл параметров WAL-G:

```
postgres@tantor:~$ cat > /var/lib/postgresql/.walgf.json <<EOF
{
  "WALG_FILE_PREFIX": "/var/lib/postgresql/tantor-se-17",
  "WALG_COMPRESSION_METHOD": "brotli",
  "WALG_DELTA_MAX_STEPS": "5",
  "PGHOST": "/var/run/postgresql/.s.PGSQL.5432",
  "PGDATA": "/var/lib/postgresql/tantor-se-server-15/main/$(cat /etc/hostname)"
}
EOF
```

Резервирование будет выполняться в директорию, на которую указывает ключ

WALG_FILE_PREFIX

2) Создайте бэкап, выполнив команду:

```
postgres@tantor:~$ wal-g --config /var/lib/postgresql/.walgf.json backup-push $PGDATA
INFO: 2035/06/25 18:38:13.859671 Couldn't find previous backup. Doing full backup.
INFO: 2035/06/25 18:38:13.884199 Calling pg_start_backup()
INFO: 2035/06/25 18:38:13.937985 Starting a new tar bundle
INFO: 2035/06/25 18:38:13.938445 Walking ...
INFO: 2035/06/25 18:38:13.939263 Starting part 1 ...
INFO: 2035/06/25 18:38:16.984992 Packing ...
INFO: 2035/06/25 18:38:16.987214 Finished writing part 1.
INFO: 2035/06/25 18:38:16.987498 Starting part 2 ...
INFO: 2035/06/25 18:38:16.987877 /global/pg_control
INFO: 2035/06/25 18:38:16.988634 Finished writing part 2.
INFO: 2035/06/25 18:38:16.988737 Calling pg_stop_backup()
INFO: 2035/06/25 18:38:17.021268 Starting part 3 ...
INFO: 2035/06/25 18:38:17.022452 backup_label
INFO: 2035/06/25 18:38:17.022772 tablespace_map
INFO: 2035/06/25 18:38:17.023600 Finished writing part 3.
INFO: 2035/06/25 18:38:17.039974 Wrote backup with name base_000000020000000000000001F
```

3) Посмотрите, какие директории и файлы были созданы:

```
postgres@tantor:~$ ls -a /var/lib/postgresql/tantor-se-17
. .. basebackups_005 data
postgres@tantor:~$ ls -a /var/lib/postgresql/tantor-se-17/basebackups_005
. base_000000020000000000000001F
.. base_000000020000000000000001F_backup_stop_sentinel.json
postgres@tantor:~$ wal-g --config /var/lib/postgresql/.walgf.json backup-push $PGDATA
INFO: 2035/06/25 18:49:13.386080 LATEST backup is: 'base_000000020000000000000001F'
INFO: 2035/06/25 18:49:13.386934 Delta backup from base_000000020000000000000001F with LSN
0/1F000028.
INFO: 2035/06/25 18:49:13.430095 Calling pg_start_backup()
INFO: 2035/06/25 18:49:13.492602 Delta backup enabled
INFO: 2035/06/25 18:49:13.492820 Starting a new tar bundle
INFO: 2035/06/25 18:49:13.493022 Walking ...
INFO: 2035/06/25 18:49:13.493596 Starting part 1 ...
INFO: 2035/06/25 18:49:13.536216 Packing ...
INFO: 2035/06/25 18:49:13.541442 Finished writing part 1.
INFO: 2035/06/25 18:49:13.541661 Starting part 2 ...
INFO: 2035/06/25 18:49:13.541839 /global/pg_control
INFO: 2035/06/25 18:49:13.543270 Finished writing part 2.
INFO: 2035/06/25 18:49:13.543529 Calling pg_stop_backup()
INFO: 2035/06/25 18:49:13.590353 Starting part 3 ...
INFO: 2035/06/25 18:49:13.590899 backup_label
INFO: 2035/06/25 18:49:13.591465 tablespace_map
```

```
INFO: 2035/06/25 18:49:13.592343 Finished writing part 3.
INFO: 2035/06/25 18:49:13.598641 Wrote backup with name
base_00000002000000000000000021_D_000000020000000000000001F
postgres@tantor:~$ ls -a /var/lib/postgresql/tantor-se-17/basebackups_005
.
..
base_0000000200000000000000001F
base_0000000200000000000000001F_backup_stop_sentinel.json
base_00000002000000000000000021_D_000000020000000000000001F
base_00000002000000000000000021_D_000000020000000000000001F
_backup_stop_sentinel.json
```

Часть 6. Остановка архивирования журналов

1) Выполните команды:

```
postgres@tantor:~ $ rm -rf /var/lib/postgresql/tantor-se-17/basebackups_005
postgres@tantor:~ $ psql -c "alter system set archive_mode = off;"
ALTER SYSTEM
postgres@tantor:~ $ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~ $ sudo systemctl start tantor-se-server-17
postgres@tantor:~$ sudo systemctl stop tantor-se-server-17-replica
```

Значение параметра `archive_mode = off` отключает резервирование WAL-сегментов.