Министерство науки и образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э.

Баумана

(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)



**МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ**

**ПО КУРСУ «АДМИНИСТРИРОВАНИЕ БАЗЫ ДАННЫХ»**

**Лабораторная работа №5** **«****Резервное копирование и репликация»**

Авторы:

Кудрявцев А.П., [kudryavtsevap@bmstu.ru](mailto:kudryavtsevap@bmstu.ru)

Фомин М.М.

Москва, 2023

Лабораторная работа №5. «Резервное копирование и репликация»

1. Резервное копирование

Существует два основных метода резервного копирования:

* логические копии (выгрузка SQL-скрипта, представляющего данные);
* трансляция журналов транзакций.

1.1. Логическое резервирование

Логическое резервирование — набор команд SQL, восстанавливающих

кластер (или базу данных, или отдельную таблицу) с нуля: создаются

необходимые объекты и наполняются данными.

Логическое резервирование — набор команд SQL, восстанавливающих

кластер (или базу данных, или отдельную таблицу) с нуля: создаются

необходимые объекты и наполняются данными.

Логическое резервирование — набор команд SQL, восстанавливающих

кластер (или базу данных, или отдельную таблицу) с нуля: создаются

необходимые объекты и наполняются данными.

Логическое резервирование — набор команд SQL, восстанавливающих

кластер (или базу данных, или отдельную таблицу) с нуля: создаются

необходимые объекты и наполняются данными.

Логическое резервирование — набор команд SQL, восстанавливающих

кластер (или базу данных, или отдельную таблицу) с нуля: создаются

необходимые объекты и наполняются данными.

Логическое резервирование — набор команд SQL, восстанавливающих кластер или базу данных или отдельную таблицу. Создаются необходимые объекты и наполняются данными.

Восстановление можно выполнить на другой версии СУБД (при наличии совместимости на уровне команд) или на другой платформе и архитектуре (не требуется двоичная совместимость). В частности, логическую резервную копию можно использовать для долговременного хранения: ее можно будет восстановить и после обновления сервера на новую версию. Однако для большой базы команды могут выполняться очень долго.

Небольшое отступление. Команда CREATE DATABASE выполняет копирование одной из существующих системных баз данных – template1, либо template0. По умолчанию копируется стандартная системная база template1. Таким образом, template1 это шаблон, на основе которого создаются новые базы. Если добавить объекты в template1, то впоследствии они будут копироваться в новые базы данных. Это позволяет внести изменения в стандартный набор объектов.

Вторая системная база template0. При инициализации она содержит те же самые объекты, что и template1, предопределённые в рамках устанавливаемой версии PostgreSQL. В template0 не следует вносить никакие изменения после инициализации кластера. Если в команде CREATE DATABASE указать в качестве шаблона template0 вместо template1, вы сможете получить «чистую» пользовательскую базу данных (в которой никаких пользовательских объектов нет, есть только системные объекты в первозданном виде), не содержащую ничего, что могло быть добавлено на месте в template1. Это особенно полезно при восстановлении дампа **pg\_dump**: скрипт выгруженного дампа должен восстанавливаться в чистую базу, чтобы он мог воссоздать нужное содержимое базы, избежав конфликтов с объектами, которые могли быть добавлены в template1.

1.1.1. Утилита pg\_dump

Для создания полноценной резервной копии базы данных используется утилита **pg\_dump**. Если не указать имя файла (-f, --file), то утилита выведет результат на консоль. А результатом является скрипт, предназначенный для **psql**, который содержит команды для создания необходимых объектов и наполнения их данными.

Дополнительными ключами утилиты можно ограничить набор объектов:

выбрать указанные таблицы, или все объекты в указанных схемах, или

наложить другие фильтры.

Чтобы восстановить объекты из резервной копии, достаточно

выполнить полученный скрипт в psql.

Дополнительными ключами утилиты можно ограничить набор объектов: выбрать указанные таблицы или все объекты в указанных схемах или наложить другие фильтры. Чтобы восстановить объекты из резервной копии, достаточно выполнить полученный скрипт в **psql**. Следует иметь в виду, что базу данных для восстановления надо создавать из шаблона template0, так как все изменения, сделанные в template1, также попадут в резервную копию.

При запуске без дополнительных параметров утилита **pg\_dump** выдает команды SQL, создающие все объекты в базе данных demo:

~$ pg\_dump -d demo > ~/Backup/demo.sql

или

~$ pg\_dump -d demo -f ~/Backup/demo.sql

В данном примере скрипты по созданию объектов и наполнению таблиц сохраняются в файле demo.sql. Важно отметить, что при таком использовании **pg\_dump** скрипт по созданию БД demo не создается. Для того, чтобы скрипт формировался надо указать параметр -C или --create:

~$ pg\_dump -d demo --create -f ~/Backup/demo.sql

Как видим, в скрипте БД формируется с использованием шаблона template0:

CREATE DATABASE demo WITH TEMPLATE = template0 …

Для того, чтобы подключиться к другому серверу, следует использовать следующие параметры подключения:

-d, --dbname=БД имя базы данных для выгрузки

-h, --host=ИМЯ имя сервера базы данных или каталог сокетов

-p, --port=ПОРТ номер порта сервера БД

-U, --username=ИМЯ имя пользователя баз данных

-w, --no-password не запрашивать пароль

-W, --password запрашивать пароль всегда (обычно не требуется)

--role=ROLENAME выполнить SET ROLE перед выгрузкой.

Восстановим БД demo из файла demo.sql (этот файл создан с ключом --create):

~$ psql -U <user> -d postgres

postgres=# \i ~/Backup/demo.sql

или

~$ psql -U <user> -d postgres -f ~/Backup/demo.sql

Утилита **pg\_dump** позволяет указать формат резервной копии. По умолчанию это plain — простые команды для psql. Формат custom (-F c, --format=custom) создает резервную копию в специальном формате, содержащем не только объекты, но и оглавление. Наличие оглавления позволяет выбирать объекты для восстановления не при создании копии, а непосредственно при восстановлении. Файл формата custom по умолчанию сжат. Для восстановления потребуется другая утилита — **pg\_restore**. Она читает файл и преобразует его в команды psql. Если не указать явно имя базы данных (в ключе -d), то команды будут выведены на консоль. Если же база данных указана — утилита соединится с этой БД и выполнит команды без участия psql.

Чтобы восстановить только часть объектов, можно воспользоваться

одним из двух подходов. Во-первых, можно ограничить объекты

аналогично тому, как они ограничиваются в pg\_dump. Вообще, утилита

pg\_restore понимает многие параметры из репертуара pg\_dump

Чтобы восстановить только часть объектов, можно воспользоваться одним из двух подходов. Во-первых, можно ограничить объекты аналогично тому, как они ограничиваются в **pg\_dump**. Вообще, утилита **pg\_restore** понимает многие параметры из репертуара pg\_dump.

Во-вторых, можно получить из оглавления список объектов, содержащихся в резервной копии (ключ --list). Затем этот список можно отредактировать вручную, удалив ненужное и передать измененный список на вход pg\_restore (ключ --use-list).

Создадим таблицу test в БД demo:

CREATE TABLE test

( id serial, name text);

INSERT INTO test( name )

VALUES('строка 1'),

('строка 2');

Выполним резервное копирование в формате custom:

~$ pg\_dump -d demo --create --format=custom -f ~/Backup/demo.cst

Удалим таблицу test из БД demo:

drop table test;

Восстановим таблицу test из резервной копии:

~$ pg\_restore -d demo --table=test ~/Backup/demo.cst

Удалим БД demo и восстановим ее целиком:

~$ pg\_restore --create -d postgres ~/Backup/demo.cst

Следует обратить внимание, что размер резервной копии demo.cst почти в четыре раза меньше, чем не custom вариант – demo.sql.

1.1.2. Утилита pg\_dumpall

Чтобы создать резервную копию всего кластера, включая роли и табличные пространства, можно воспользоваться утилитой **pg\_dumpall**. Поскольку **pg\_dumpall** требуется доступ ко всем объектам всех БД, имеет смысл запускать ее от имени суперпользователя. Утилита по очереди подключается к каждой БД кластера и выгружает информацию с помощью **pg\_dump**. Кроме того, она сохраняет и данные, относящиеся к кластеру в целом. Чтобы начать работу, утилите требуется подключиться хотя бы к какой-то базе данных. По умолчанию выбирается postgres или template1, но можно указать и другую. Результатом работы **pg\_dumpall** является скрипт для psql.

При создании обычной копии с помощью **pg\_dump** базы данных сохраняются все права, но не команды CREATE USER. Эти глобальные данные не включаются в обычную копию, их может извлечь только **pg\_dumpall**. Если нужны только глобальные данные, запустите pg\_dumpall с ключом -g (--globals-only).

Но поскольку для копирования кластера надо иметь доступ ко всем БД, мы выполняем **pg\_dumpall** под суперпользовательской ролью.

pg\_dumpall -U <super user> -f ~/Backup/demoall.sql

1.2. Физическое резервирование

Физическое резервирование подразумевает копирование всех файлов, относящихся к кластеру БД, то есть создание полной двоичной копии. Копирование файлов работает быстрее, чем выгрузка SQL-команд при логическом резервировании; запустить сервер из созданной физической копии — дело нескольких минут в отличие от восстановления из логической копии. Кроме того, нет необходимости заново собирать статистику — она также восстанавливается из копии. Но есть и минусы. Из физической резервной копии можно восстановить систему только на совместимой платформе (та же ОС, та же разрядность, тот же порядок байтов в представлении чисел и т. п.) и только на той же основной версии PostgreSQL. Кроме того, невозможно сделать физическую копию отдельных баз данных кластера, возможно копирование только всего кластера целиком.

1.2.1. Настройка резервирования WAL

Создадим папку /Wal и изменим у нее собственника на postgres:

sudo mkdir /Wal

sudo chown postgres.postgres /Wal

Внесем несколько изменений в файл postgresql.conf. Большинство параметров установлено по умолчанию так, чтобы выполнять архивирование и репликацию, но не все:

wal\_level = replica

max\_wal\_senders = 10

Параметр wal\_level говорит, что сервер должен записывать в журнал транзакций достаточно информации.

Параметр max\_wal\_senders задаёт максимально допустимое число одновременных подключений ведомых серверов или клиентов потокового копирования (т. е. максимальное количество одновременно работающих процессов передачи WAL).

Включим режим архивирования и установим команду копирования заполненных сегментов журнала.

archive\_mode = on

В archive\_command задаем копирование файлов в папку Backup:

archive\_command='test ! -f / Wal/%f && cp %p /Wal/%f'

Postgres замещает параметр %p файлом архива, а %f – именем файла.

Сначала проверяется наличие файла с указанным именем в архиве, а затем в случае отсутствия копируется.

Сохраняем файл postgresql.conf.

Изменим файл pg\_hba.conf. Это необходимо, при использовании утилитой pg\_basebackup – средством создания резервных копий базы данных.

# Allow replication connections from localhost, by a user with the

# replication privilege.

local replication all trust

Перезапускаем сервер PostgreSQL.

1.2.2. Создание базовой резервной копии

Архивацию файлов WAL настроили, создадим первую резервную копию. Имея ее и последующие файлы WAL, мы сможем восстановить состояние базы данных на любой момент времени.

Создадим папку /Base и сделаем ее собственником postgres:

sudo mkdir /Base

sudo chown postgres.postgres /Base

Для создания начальной копии воспользуемся командой pg\_basebackup:

sudo pg\_basebackup -U postgres -D /Base -h localhost --checkpoint=fast   
 --wal-method=stream

У этой команды четыре параметра.

* -D: где должна находиться базовая резервная копия? PostgreSQL требует, чтобы это был пустой каталог. По завершении копирования в нем окажется копия каталога data на сервере.
* -h: IP-адрес или имя главного сервера (источника), который мы собираемся скопировать.
* --checkpoint=fast: обычно pg\_basebackup ждет, пока главный сервер не создаст контрольную точку, поскольку процесс восстановления должен с чего-то начинаться. Чтобы не ждать, а это может затянуться на десятки минут, принудительно ускорим запись контрольной точки.
* --wal-method=stream: по умолчанию pg\_basebackup подключается к главному серверу и начинает копировать файлы. Но имейте в виду, что пока происходит копирование, эти файлы могут модифицироваться. Поэтому данные в резервной копии могут оказаться несогласованными. Эту несогласованность можно устранить в процессе восстановления с помощью журнала транзакций. Но сама резервная копия необязательно согласована. Параметр --wal-method=stream позволяет создать согласованную резервную копию, из которой можно восстановить базу данных, не воспроизводя журнал транзакций. Это удобно, если нам нужно всего лишь клонировать экземпляр, а восстановление на определенный момент времени ни к чему.

2. Восстановление БД

2.1. Копирование данных

В процессе работы системы может произойти авария, и нам придется восстанавливать базу. Сымитируем разрушение БД. Для этого остановим PostgreSQL и удалим (или переместим) данные. Данные обычно находятся в папке:

/var/lib/postgresql/<№ версии>/main

Расположение данных можно выяснить, выполнив запрос:

SELECT \* FROM pg\_file\_settings;

Здесь представление pg\_file\_settings предоставляет сводную информацию о содержимом файлов конфигурации сервера. Значение для data\_directory является искомой папкой.

Сигналом при запуске PostgreSQL о том, что выполняется восстановление БД, является наличие файла **recovery.signal.** Этот файл создается вручную. Содержимое файла игнорируется. Если **recovery.signal** отсутствует, то PostgreSQL считает, что он выполняет автоматическое восстановление после сбоя. Создадим **recovery.signal** в папке/Base (резервная копия):

sudo -i

touch /Base/**recovery.signal**

Перенесем данные в некую папку, пусть это будет папка /WWW.

cd /

mkdir /WWW

cd /var/lib/postgresql/<№ версии>/main

mv \* /WWW

Перенесем резервную копию в папку данных PostgreSQL (в данный момент текущей папкой является - /var/lib/postgresql/<№ версии>/main):

mv /Base/\* ./

Всем файлам и папкам назначаем собственника postgres:

chown -R postgres.postgres main

2.2. Редактирование postgres.conf

Внесем изменения в файл postgres.conf. Задать параметр **restore\_command**, который определяет команду, «обратную» команде архивирования archive\_command. Она должна скопировать файл из архива в каталог pg\_wall. В случае при возникновении ошибки оболочки (например, если команда не найдена), процесс восстановления будет остановлен, и сервер не запустится.

restore\_command = 'cp /Wal/%f %p'

Зададим цель – параметр **recover y\_target\_action**, который позволяет выбрать состояние, в котором окажется сервер по окончании восстановления. Вариант по умолчанию — pause, что означает приостановку восстановления. Второй вариант, promote, означает, что процесс восстановления завершится, и сервер начнёт принимать подключения. Наконец, с вариантом shutdown сервер остановится, как только цель восстановления будет достигнута.

Вариант pause позволяет выполнить запросы к базе данных и убедиться в том, что достигнутая цель оказалась желаемой точкой восстановления. Для снятия с паузы нужно вызвать pg\_wal\_replay\_resume(), что в итоге приведёт к завершению восстановления. Если же окажется, что мы ещё не достигли желаемой точки восстановления, нужно остановить сервер, установить более позднюю цель и перезапустить сервер для продолжения восстановления.

Вариант shutdown полезен для получения готового экземпляра сервера в желаемой точке. При этом данный экземпляр сможет воспроизводить дополнительные записи WAL (а при перезапуске ему придётся воспроизводить записи WAL после последней контрольной точки).

Заметьте, что так как recovery.signal не переименовывается, когда в recovery\_target\_action выбран вариант shutdown, при последующем запуске будет происходить немедленная остановка, пока вы не измените конфигурацию или не удалите файл recovery.signal вручную.

recovery\_target\_action = 'promote'

После сохранения файла postgres.conf запустим PostgreSQL:

sudo systemctl start postgresql

После успешного восстановления файл **recovery.signal** удаляется.

2.3. Целевая точка

Мы не задали цель восстановления (один из параметров   
recovery\_t\_arget\_\* в файле postgres.conf), поэтому БД будут восстановлены максимально близко к моменту сбоя. Однако процесс восстановления можно остановить и в любой другой момент, указав целевую точку.

Параметр recovery\_target = 'immediate' остановит восстановление, как только будет достигнута согласованность. Фактически, это эквивалентно восстановлению из базовой копии без архива. Параметр recovery\_target\_name позволяет указать именованную точку восстановления, созданную ранее с помощью функции pg\_create\_restore\_point(). Это полезно, если заранее известно, что может потребоваться восстановление. Параметр recovery\_target\_time позволяет указать произвольное время (timestamp), recovery\_target\_xid — произвольную транзакцию, recovery\_target\_lsn —указывает LSN позиции в журнале предзаписи, до которой должно выполняться восстановление. Параметр recovery\_target\_inclusive указывает на необходимость остановки сразу после (on) либо до (off) достижения целевой точки.

2.4. Очистка архива

17

Очистка архива

17

Очистка архива

Архив со временем разрастается, поэтому требуется периодическая очистка от ненужных файлов. Для очистки можно воспользоваться утилитой **pg\_archivecleanup**. Утилита pg\_archivecleanup предназначена для использования в качестве archive\_cleanup\_command для удаления старых файлов WAL на резервном сервере или восстановлении БД после сбоя. **pg\_archivecleanup** можно использовать и как отдельную программу для выполнения тех же действий. Чтобы использовать pg\_archivecleanup на резервном сервере, добавьте эту строку в файл конфигурации postgresql.conf:

archive\_cleanup\_command = 'pg\_archivecleanup ***расположение\_архива*** %r'

В команде можно использовать комбинацию %r, которая заменится на имя файла, содержащего запись о последней выполненной контрольной точке.

При отдельном использовании этой утилиты

pg\_archivecleanup [***парам***] ***расположение\_архива*** ***старейший\_сохраняемый\_файл***

из каталога ***расположение\_архива*** будут удалены все файлы WAL, логически предшествующие файлу ***старейший\_сохраняемый\_файл***. В этом режиме, если вы укажете имя файла с расширением .partial или .backup, ***старейший\_сохраняемый\_файл*** будет определяться по имени без расширения. Благодаря такой интерпретации расширения .backup будут корректно удалены все файлы WAL, заархивированные до определённой базовой копии.

Например, после выполнения резервного копирования в каталоге /Wal имеем следующие файлы:

-rw------- 1 postgres postgres 16777216 фев 5 11:01 0000000200000002000000C2

-rw------- 1 postgres postgres 16777216 фев 5 14:40 0000000200000002000000C3

-rw------- 1 postgres postgres 16777216 фев 5 14:42 0000000200000002000000C4

-rw------- 1 postgres postgres 16777216 фев 5 14:42 0000000200000002000000C5

-rw------- 1 postgres postgres 341 фев 5 14:42 0000000200000002000000C5.00000028.backup

Файлы WAL, которые старше файла 0000000200000002000000C5.00000028.backup не нужны, удалим:

pg\_archivecleanup -d /Wal 0000000200000002000000C5.00000028.backup

pg\_archivecleanup: keeping WAL file "/Wal/0000000200000002000000C5" and later

pg\_archivecleanup: removing file "/Wal/0000000200000002000000C3"

pg\_archivecleanup: removing file "/Wal/0000000200000002000000C2"

pg\_archivecleanup: removing file "/Wal/0000000200000002000000C4"

3. Асинхронная репликация

Репликация БД – это процесс синхронизации данных в режиме реального времени между несколькими серверами БД. Репликация позволяет распределить нагрузку на серверы, обеспечить отказоустойчивость и повысить доступность данных. Основная цель репликации БД — обеспечить согласованность данных между репликами. В процессе репликации все изменения, вносимые в основную БД (мастер), автоматически передаются на реплики.

Мастер передает на реплику журнальные записи, а реплика применяет эти записи к своим файлам данных. Применение происходит чисто механически. Поэтому критична двоичная совместимость между серверами — они должны работать на одной и той же программно-аппаратной платформе и на них должны быть запущены одинаковые основные версии PostgreSQL.

Физическая репликация может быть как синхронной так и асинхронной. При асинхронной репликации после завершения транзакции данные не сразу передаются на реплики, а спустя некоторое время, а не в той же транзакции.

Основное преимущество этого типа репликации данных заключается в том, что в случае сбоя определенного сервера базы данных процесс репликации может продолжаться, и все транзакции в системе репликации будут зафиксированы.

При синхронной репликации завершение операции commit означает, что все журнальные записи, относящиеся к данной транзакции, переданы на реплику. Важно понимать, что получение репликой журнала не означает применения изменений к данным. Синхронная репликация данных подходит для приложений, требующих немедленной синхронизации данных. Однако синхронная репликация данных требует, чтобы все аппаратные компоненты и сети в системе репликации были доступны в любое время.

Для того чтобы организовать репликацию между двумя серверами PostgreSQL, установим Docker и создадим два PostgreSQL контейнера.

3.1. Docker

Нашей целью является организация репликации между двумя контейнерами PostgreSQL. В качестве основного сервера будем использовать первый контейнер (postgres\_cnt), а в качестве реплики или ведомого – второй контейнер (postgres\_cnt2). Создадим пользователя user\_rep с правами репликации на основном сервере, от имени которого на ведомом сервере будет выполняться соединение к основному для репликации.

3.1.1. Инсталляция Docker Engine

Перейдите на официальный сайт

<https://docs.docker.com/engine/install/>

и установите Docker Engine. В конце страницы есть ссылка на дополнительные шаги:

<https://docs.docker.com/engine/install/linux-postinstall/>

После завершения установки программы перезагрузитесь.

Если у вас установлен VSCode, то проинсталлируйте плагины поддержки Docker и, тем самым, получите альтернативу командной строке.

3.1.2. Файл docker-compose.yaml

Создадим папки, в которых будут находится наши файлы и данные БД. ~/Projects/Postgres/Data/PostgresCnt

~/Projects/Postgres/Data/PostgresCnt2

Названия и расположения папок не являются обязательными, но т. к. они используются в файле docker-compose.yaml и при организации репликации будьте внимательными при их изменении.

В папке Postgres создадим файл docker-compose.yaml:

version: '3.9'

services:

postgres\_cnt:

container\_name: postgres\_cnt

image: postgres:14.10-alpine

volumes:

- '~/Projects/Postgres/Data/PostgresCnt:/var/lib/postgresql/data'

ports:

- '5433:5432'

environment:

- POSTGRES\_PASSWORD=12345

user: '1000:1000'

restart: unless-stopped

networks:

- dbnet

postgres\_cnt2:

container\_name: postgres\_cnt2

image: postgres:14.10-alpine

volumes:

- '~/Projects/Postgres/Data/PostgresCnt2:/var/lib/postgresql/data'

ports:

- '5434:5432'

environment:

- POSTGRES\_PASSWORD=12345

user: '1000:1000'

networks:

- dbnet

restart: unless-stopped

networks:

dbnet:

name: db\_net

driver: bridge

Разберем содержимое файла. service — это абстрактное определение вычислительного ресурса внутри приложения, являющегося элементом верхнего уровня. Определение службы содержит конфигурацию, которая применяется к каждому контейнеру службы.

Здесь описаны две службы postgres\_cnt и postgres\_cnt2, и соответственно, два одноименных контейнера (название контейнеров может отличаться от названия службы). Название контейнеров используется для их идентификации при работе с ними. Контейнер — это уже развернутое и запущенное приложение. Контейнер можно сравнить с уже установленной и работающей программой на ПК.

image – это неизменяемый образ, из которого разворачивается контейнер. Его можно рассматривать как набор файлов, необходимых для запуска и работы приложения на другом хосте. В качестве образа используется официальный образ PostgreSQL 14, взятый с официального сайта <https://hub.docker.com/>. При создании контейнеров будет скачен один образ с сайта и на основе этого образа будут создаваться контейнеры. Если не удалять образы, то в дальнейшем они скачиваться не будут.

Контейнер — это изолированный комплекс программ и данных. При удалении контейнера все настройки и данные, естественно, тоже удалятся. Чтобы этого избежать и сохранить БД и файлы конфигурации, необходимо «связать» (mapping) нашу папку, внешнюю по отношению к контейнеру с папкой данных в контейнере. Эта привязка выполняется ключом volumes. При первом запуске в эту папку скопируются БД по умолчанию.

По умолчанию, когда вы создаете или запускаете контейнер с помощью Docker, контейнер не раскрывает ни один из своих портов внешнему миру. Поэтому, чтобы сделать порт доступным для служб за пределами Docker, необходимо использовать ключевое слово ports. Это создает правило брандмауэра на хосте, сопоставляющее порт контейнера порту хоста Docker с внешним миром. В нашем случае, доступ к БД для первого контейнера будет выполняться через порт 5433, а для второго – 5434. (Порт 5432 занят, установленным на компьютере PostgreSQL.)

При создании контейнеров PostgreSQL создаются БД postgres и суперпользователь postgres. Переменная окружения POSTGRES\_PASSWORD задает пароль для пользователя postgres (введите свой пароль).

По умолчанию Docker-контейнер запускается от имени пользователя root. Поскольку контейнеры Docker используют одно и то же ядро с хост-системой, запуск от имени пользователя root подвергает всю систему потенциальным атакам. В настоящий момент вопросы безопасности и удобство работы с БД у нас совпадают. Данные мы разместили в наших домашних папках ~/Projects/Postgres/Data/PostgresCnt (PostgresCnt2), т. е. являемся их собственникам. Было бы неплохо остаться собственником вложенных папок и файлов, созданных Docker. Для этого надо указать свое user: uid:gid.

Для того, чтобы узнать id введите команду:

:~$ id <user>

или

:~$ id $USER

uid=1000(alex) gid=1000(alex) groups=1000(alex),…

Создадим собственную пользовательскую сеть db\_net и подключим контейнеры к одной сети. После подключения к определяемой пользователем сети контейнеры могут взаимодействовать друг с другом, используя IP-адреса контейнеров или имена контейнеров.

Строкой restart: unless-stopped мы указываем, чтобы контейнеры запускались при включении компьютера, если они работали при выключении компьютера.

3.1.3. Запуск docker-compose.yaml

Перейдем в папку ~/Projects/Postgres, создадим, и запустим контейнеры:

cd ~/Projects/Postgres

docker compose up -d

Docker в находит в папке docker-compose.yaml и приступает к созданию контейнеров. (Можно не переходить в папку, которая содержит файл yaml, и имя этого файла может быть любое. Но тогда надо будет прописать путь к этому файлу при вызове docker compose. Использование опций docker compose оставляем для самостоятельного изучения.) Сначала выполняется скачивание образов контейнером, в нашем случае один образ, это займет какое-то время. После этого контейнеры запускаются. Для того чтобы увидеть запущенные контейнеры выполните команду:

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

f02abe7d9123 postgres:14.10-alpine "docker-entrypoint.s…" 3 days ago Up 9 minutes 0.0.0.0:5434->5432/tcp, :::5434->5432/tcp postgres\_cnt2

a70e5dc3a6bc postgres:14.10-alpine "docker-entrypoint.s…" 3 days ago Up 9 minutes 0.0.0.0:5433->5432/tcp, :::5433->5432/tcp postgres\_cnt

Как видим статус Up, значит контейнеры работают. Но для того, чтобы убедиться, что все в порядке, надо посмотреть логи:

docker compose logs

3.1.4. Команды Docker

Ниже перечислен минимальный перечень команд, необходимый для выполнения лабораторной работы.

Список команд:

docker --help

docker compose --help

Активные контейнеры:

docker ps

Все, включая не активные, контейнеры:

docker ps -a

Запустить контейнер:

docker start <id/имя контейнера>

Остановить контейнер:

docker stop <id/имя контейнера>

Удалить контейнер:

docker rm <id/имя контейнера>

Удаление всех остановленных контейнеров:

docker container prune

Список образов:

docker images

Удалить образ:

docker rmi <id image/имя образа>

Для того чтобы «войти» в запущенный контейнер и что-то сделать на стороне контейнера, необходимо использовать команду exec в комбинации с опциями. Например, войдем в контейнер PostgreSQL:

docker exec -it postgres\_cnt bash

Опция i включает интерактивный режим, а – t открывает терминал. После имени контейнера надо ввести команду. В примере введена командная оболочка bash. Но не во всех контейнерах доступна команда bash. В нашем случае образом является postgres:14.10-alpine, где alpine есть наименьшая ОС Linux. После входа в контейнер мы получаем систему команд для работы с файлами и папками, поиском, настройкой окружения и т.д. Но надо держать в уме, что все изменения, которые не фиксируются в связанных папках (volumes) с удалением контейнера пропадут. Для выхода из контейнера выполните команде exit.

Создать и запустить контейнеры с применением файла docker-compose.yaml в фоновом режиме (-d):

docker compose up -d

Остановка всех контейнеров:

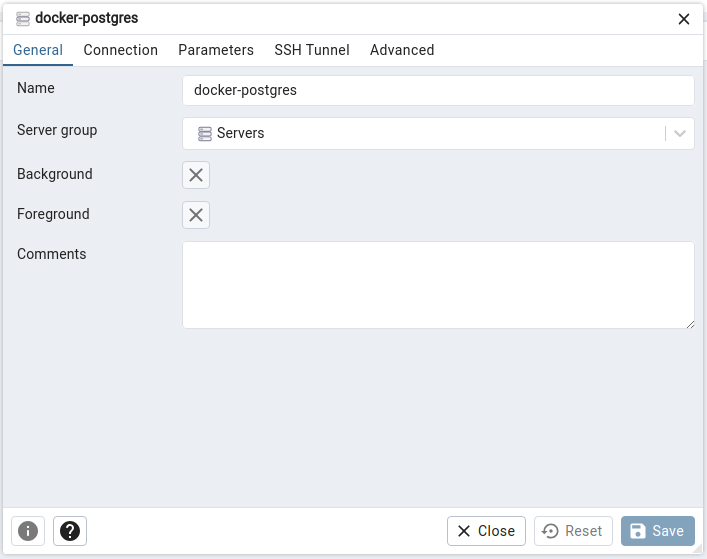
docker compose stop

Остановка и удаление всех контейнеров и docker-сетей:

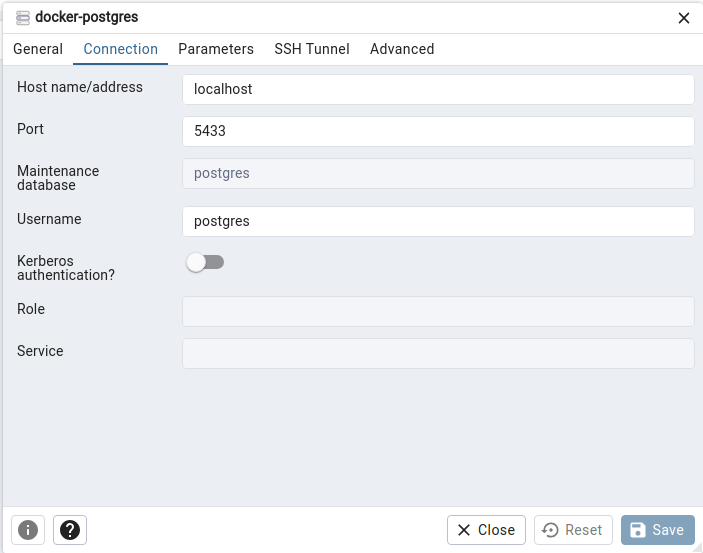
docker compose down

3.2. Регистрация в pgAdmin

Запустим pgAdmin и нажмем правую кнопку мышки на верхнем узле Servers из меню команду Register/Server.



Название можно ввести любое.



Для первого контейнера вводим «внешний» порт 5433. Имя пользователя по умолчанию – postgres и пароль такой же, как и в файле docker-compose.yaml.

Аналогично зарегистрируйте второй контейнер, для которого надо указать порт 5434.

3.3. Настройка репликации

3.3.1. Изменения для первого контейнера postgres\_cnt

Создадим пользователя user\_rep с правами выполнения репликации, который будет подключаться со стороны второго сервера. Для этого необходимо войти в контейнер с правами root, а в контейнере войти с правами postgres, т. к. на данный момент пользователь postgres имеет права на создание ролей:

docker exec -it –user=root postgres\_cnt bash

su postgres

createuser --replication -P user\_rep

Опция -P инициирует запрос на ввод пароля, который необходимо ввести два раза.

Подключаемся к БД и создаем слот ‘replica’:

psql -U postgres

SELECT pg\_create\_physical\_replication\_slot('replica');

Посмотрим на созданный слот:

SELECT \* FROM pg\_replication\_slots \gx

-[ RECORD 1 ]-------+---------

slot\_name | replica

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 |

two\_phase | f

Вначале слот не инициализирован (restart\_lsn и wal\_status пустые).

Отсоединяемся от БД:

exit

Выходим из-под пользователя postgres:

exit

Выходим из контейнера:

exit

Открываем конфигурационный файл postgresql.conf у контейнера postgres\_cnt с помощью nano либо VSCode:

nano /var/Data/PostgresCnt/postgresql.conf

Проверяем следующие параметры (они должны быть установлены по умолчанию):

wal\_level = replica

max\_wal\_senders = 10

max\_replication\_slots = 10

hot\_standby = on

Параметр hot\_standby\_feedback в данном случае можно оставить off, он предназначен для того, чтобы дать ведомому серверу завершить начатую транзакцию.

Поясним назначение перечисленных параметров (см. документацию):

* ***wal\_level***указывает, сколько информации записывается в WAL (журнал операций, который используется для репликации). Значение **replica** указывает на необходимость записывать только данные для поддержки архивирования WAL и репликации.
* **max\_wal\_senders** — количество планируемых ведомых серверов;
* **max\_replication\_slots** — максимальное число слотов репликации;
* **hot\_standby** — определяет, можно или нет подключаться к postgresql для выполнения запросов в процессе восстановления;
* **hot\_standby\_feedback** — определяет, будет или нет ведомый сервер сообщать мастеру о запросах, которые он выполняет.

Для того, чтобы ведомый сервер получал реплики, необходимо прописать его IP адрес в файле pg\_hba.conf основного сервера. Выясним, какую подсеть используют контейнеры с postgresql:

docker network inspect db\_net | grep Subnet

В моем случае:

"Subnet": "172.27.0.0/16"

Отредактируем файл pg\_hba.conf

nano /var/Data/PostgresCnt/pg\_hba.conf

И добавляем строку после остальных «host replication»:

host replication all 172.27.0.0/16 md5

Перезапустим докер контейнер:

docker restart postgre\_cnt

3.3.2. Изменения для контейнера postgres\_cnt2

На ведомом сервере перед тем, как выполнить копирование, необходимо очистить паку с данными. А затем сразу выполнить бэкап. Промежуток между этими операциями должен быть небольшой (несколько секунд). Это обусловлено тем, что PostgreSQL в контейнере, обнаружив пустую папку, восстановит БД по умолчанию. Можно было остановить второй контейнер и выполнить бэкап на хосте, но тогда пропадет имитация работы на ведомом сервере.

Заходим во второй контейнер:

docker exec -it postgres\_cnt2 bash

Удалим все данные:

rm -r /var/lib/postgresql/data/\*

Выполним (надо поспешить) :

pg\_basebackup --host=postgres\_cnt --username=user\_rep   
 --pgdata=/var/lib/postgresql/data   
 --wal-method=stream --write-recovery-conf

--slot=replica

Утилита pg\_basebackup предназначена для создания базовых копий работающего кластера баз данных PostgreSQL. Процедура создания копии не влияет на работу других клиентов базы. pg\_basebackup создаёт точную копию файлов кластера, автоматически включая режим резервного копирования и завершая его. Такие резервные копии всегда создаются для кластера целиком.

В качестве хоста указано имя первого контейнера postgres\_cnt. При задании параметра --write-recovery-conf или -R создается файл [standby.signal](https://postgrespro.ru/docs/postgrespro/15/warm-standby#FILE-STANDBY-SIGNAL) и добавляются параметры конфигурации в файл postgresql.auto.conf в целевой папке. Это упрощает настройку ведомого сервера при восстановлении копии. А в файл postgresql.auto.conf будут записаны параметры соединения и слот репликации, если его использует pg\_basebackup, так что впоследствии при потоковой репликации будут использоваться те же параметры.

3.3.3 Проверка

Войдем в первый контейнер и запустить psql. Проверим состояние слота:

postgres=# SELECT \* FROM pg\_replication\_slots \gx

-[ RECORD 1 ]-------+-----------

slot\_name | replica

plugin |

slot\_type | physical

datoid |

database |

temporary | f

active | t

active\_pid | 712

xmin |

catalog\_xmin |

restart\_lsn | 0/150003D0

confirmed\_flush\_lsn |

wal\_status | reserved

safe\_wal\_size |

two\_phase | f

После выполнения резервной копии слот инициализировался, и мастер теперь хранит все файлы журнала с начала копирования (restart\_lsn, wal\_status).

Представление pg\_stat\_replication для каждого процесса-передатчика WAL будет содержать по одной строке со статистикой о репликации на ведомый сервер, к которому подключён этот процесс.

postgres=# select \* from pg\_stat\_replication \gx

-[ RECORD 1 ]----+------------------------------

pid | 712

usesysid | 16537

usename | user\_rep

application\_name | walreceiver

client\_addr | 172.27.0.8

client\_hostname |

client\_port | 36518

backend\_start | 2024-02-09 09:49:16.436271+00

backend\_xmin |

state | streaming

sent\_lsn | 0/150003D0

write\_lsn | 0/150003D0

flush\_lsn | 0/150003D0

replay\_lsn | 0/150003D0

write\_lag |

flush\_lag |

replay\_lag |

sync\_priority | 0

sync\_state | async

reply\_time | 2024-02-09 10:41:31.900756+00

Представление pg\_stat\_wal\_receiver дает нам статистику получения WAL от сервера, с которым работает ведомый сервер. Зайдем во второй контейнер и запустим psql.

postgres=# select \* from pg\_stat\_wal\_receiver \gx

-[ RECORD 1 ]---------+------------------------------------------------

pid | 18

status | streaming

receive\_start\_lsn | 0/15000000

receive\_start\_tli | 1

written\_lsn | 0/150003D0

flushed\_lsn | 0/150003D0

received\_tli | 1

last\_msg\_send\_time | 2024-02-09 10:52:52.842375+00

last\_msg\_receipt\_time | 2024-02-09 10:52:52.842402+00

latest\_end\_lsn | 0/150003D0

latest\_end\_time | 2024-02-09 09:58:48.033964+00

slot\_name | replica

sender\_host | postgres\_cnt

sender\_port | 5432

conninfo | user=user\_rep password=\*\*\*\*\*\*\*\* channel\_binding=prefer dbname=replication host=postgres\_cnt port=5432 fallback\_application\_name=walreceiver sslmode=prefer sslcompression=0 sslsni=1 ssl\_min\_protocol\_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres target\_session\_attrs=any

3.3.4. Остановка репликации

В случае аварии на основном сервере ведомый сервер можно сделать основным, введя команду:

pg\_ctl promote

Если вы используете сервера в отладочных целях, то после прекращения репликации удалите слот на основном сервере, т. к. там будут собираться не переданные файлы WAL:

SELECT pg\_drop\_replication\_slot('slot\_name');

4. Вопросы для самостоятельной проработки

1. Для чего служат БД template0, template1.
2. Какую БД template0, template1 выбирать для восстановления.
3. Ключи pg\_dump.
4. Ключи pg\_dumpall.
5. Ключи pg\_ctl.
6. Ключи pg\_restore.
7. Журнал WAL, для чего предназначен и когда формируется.
8. Целевые точки.
9. Ключи pg\_archivecleanup.
10. Ключи pg\_basebackup.
11. Последовательность действий для организации резервного копирования (логического и физического).
12. Последовательность действий для организации асинхронной репликации.

5. Практическое задание

1. Выполнить логическое резервное копирование и восстановление БД.
2. Выполнить восстановление только одной таблицы.
3. Выполнить логическое резервное копирование и восстановление всего кластера.
4. Выполнить физическое резервное копирование и восстановление.
5. Организовать асинхронную репликацию с использованием Docker. Добавить в первый контейнер БД demo. Создать таблицу и внести в нее данные. Убедиться, что в реплике изменения отразились

6. Список использованных источников

1. Документация PostgreSQL https://postgrespro.ru/docs
2. Документация Docker <https://docs.docker.com/manuals/>
3. PostgreSQL 11. Мастерство разработки. Ганс-Юрген Шениг / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2019.