

PostgreSQL 13.

Тюнинг,

Kubernetes,

облака.

Второе издание.

УДК 004.4/.6
ББК 32.97
А81

А81 Аристов Евгений. PostgreSQL 13. Тюнинг, Kubernetes, облака. - М.: ООО «Сам Полиграфист», 2021. - 492 с. Доработанное издание.

ISBN

В данной книге подробно изучается внутреннее устройство PostgreSQL 13, начиная с вариантов установки, физического устройства этой системы управления базами данных, заканчивая бэкапами и репликацией, а также разбираются кластерные и облачные решения на основе PostgreSQL.

Книга написана понятным языком с использованием подробных примеров. Исходные коды и ссылки, используемые в книге, выложены на github.

Главы логически выстроены по пути усложнения материала и позволяют разобраться, как правильно развернуть кластер PostgreSQL с нуля на отдельных серверах, в виртуальном и облачном исполнении. Подробно рассматриваются настройки, отвечающие за производительность и варианты их тюнинга.

Отдельно выделены несколько глав, где рассматриваются варианты оптимизации с использованием различных механизмов: индексы, секционирование, джойны и другое.

Книга будет интересна и полезна широкой аудитории: разработчикам, администраторам баз данных, архитекторам программного обеспечения, в том числе микросервисной архитектуры.

Это вторая редакция – уменьшено количество теоретических материалов, добавлено немного практики и ссылок на дополнительное изучение.

Тираж 100 экз. Заказ № ____.

Отпечатано в типографии «OneBook.ru» ООО «Сам Полиграфист»
129090 г. Москва, Протопоповский пер., 6
www.onebook.ru

© Аристов Е.Н., 2021.

Оглавление

Об авторе	4
1. Установка PostgreSQL 13	156
2. Физический уровень	23
3. Работа с консольной утилитой <code>psql</code>	42
4. ACID && MVCC. Vacuum и autovacuum	48
5. Уровни изоляции транзакций	63
6. Логический уровень	73
7. Работа с правами пользователя	87
8. Журналы	102
9. Блокировки	131
10. Настройка PostgreSQL	157
11. Работа с большим объёмом реальных данных	178
12. Виды индексов. Работа с индексами и оптимизация запросов	194
13. Различные виды join'ов. Применение и оптимизация	215
14. Сбор и использование статистики	240
15. Оптимизация производительности. Профилирование. Мониторинг	252
16. Секционирование	267
17. Резервное копирование и восстановление	287
18. Виды и устройство репликации в PostgreSQL	309
19. PostgreSQL и Google Kubernetes Engine	330
20. Кластеры высокой доступности для PostgreSQL	364
21. Кластеры для горизонтального масштабирования PostgreSQL	398
22. PostgreSQL и Google Cloud Platform	432
23. PostgreSQL и AWS	453
24. PostgreSQL и Azure	469
25. PostgreSQL и Яндекс Облако	481
Послесловие от автора	492

Об авторе

Для начала, хотелось бы поблагодарить за то, что вы выбрали эту книгу. Я в IT уже больше 20 лет. Начинал, как и многие когда-то с ZX Spectrum. Написал свою первую игру наподобие текстового Dictator. Тогда все хранилось на простых аудиокассетах, а язык был Basic... Потом IBM 8086. Хотя кто уже помнит те времена... Дальше DOS, первый Windows 3.11, - профильный университет, базы данных большие и маленькие, сотни проектов, - виртуализация, кластеризация и highload. В какой-то момент понял, что хочу учить людей, нести светлое и доброе в наш мир. И вот я больше года читаю свои авторские курсы по Постгресу в «OTUS Онлайн-образование»¹ и «УЦ Специалист» при МГТУ им Н.Э. Баумана².

За это время понял, насколько не хватает хорошей книги по лучшим методикам и возможностям самой популярной СУБД³ PostgreSQL. В этой книге будут описаны практические рекомендации, проведён глубокий анализ внутреннего устройства для понимания принципов работы системы и возможностей её тонкой настройки. Ну и куда без самого популярного оркестратора контейнеризации k8s⁴ и облаков. Конечно рассмотрим и различные виды кластеризации Постгреса.

Ссылки из данной книги будут расположены на github⁵ для удобства перехода. В репозитории ссылки будут разбиты по номерам тем, номер ссылки будет в квадратных скобках [1]. Я всегда готов к сотрудничеству - об этом можно посмотреть на моём личном сайте: <https://aristov.tech>.

Спасибо за поддержку Илье @JorianR, который и положил начало моей преподавательской карьеры. Отличный комьюнити-менеджер. Спасибо Алексею Цыкунову @erlong15. Мой наставник, Профессионал с большой буквы, опытнейший DBA⁶ и просто замечательный человек. Спасибо любимой жене Светлане @asveta за поддержку. Спасибо редактору книги @alandreei, который заставил меня плакать, переделывая финальный вариант раз пять. Спасибо Сергею Краснову @SerjReds - приложил руку к моему долгому пути к преподаванию. Ну, и конечно же, любимой дочке Анастасии, это её рисунок вы видите на обложке книги.

Желаю вам получить удовольствие от прочтения книги и удачи в карьере.

¹ OTUS URL: <https://otus.ru/> (дата обращения 15.02.2021)

² УЦ Специалист URL: <https://www.specialist.ru/> (дата обращения 15.02.2021)

³ Система Управления Базами Данных

⁴ Kubernetes

⁵ Мой github URL: https://github.com/aeuge/Postgres13book_v2 (дата обращения 15.02.2021)

⁶ DataBase Architect/Administrator

1. Установка PostgreSQL 13

Давайте рассмотрим основные варианты установки СУБД. Их можно разделить как по видам ОС⁷, так и по способу установки: на конкретной ОС, docker⁸, docker-compose⁹, в k8s различными способами: через манифесты или пакетный менеджер helm¹⁰. Также в этой главе мы рассмотрим вариант использования Постгреса как DBaaS¹¹ в облаке Google. Более подробно облако Google и остальные облака мы рассмотрим в соответствующих главах.

Начнём с самого общепринятого - установка СУБД на конкретной ОС. В текущих реалиях есть три стандартных ОС: Windows, Linux, MacOS X. Рекомендую использовать Linux, но в принципе работать будет на любой ОС. Здесь и далее будут ссылки на сайт с официальной документацией PostgreSQL Global Development Group <https://www.postgresql.org/>. Ссылка на страницу с дистрибутивами¹².

Итак, **Windows**. Постгрес выпускается для следующих 64-битных версий Windows:

- Windows 8.1, 10
- Windows Server 2008 R2 и новее

Опять есть два варианта:

- установка в графическом интерактивном режиме
- в режиме командной строки

В первом варианте всё просто, скачиваем установочный дистрибутив и, ответив на несколько несложных вопросов и нажимая кнопку далее, мы установим СУБД на компьютер. Ссылка на дистрибутив для Windows¹³.

В режиме командной строки мы можем сразу сами задать ряд параметров, таких как путь установки, каталог с данными, пароль суперпользователя и так далее. Для замены стандартных параметров, вы можете использовать INI-файл и указать там один или несколько параметров.

Но в целом, установка Постгреса на Windows в продакшн-системах не рекомендуется. В тренировочных целях, в принципе, можно.

⁷ Операционная система

⁸ Docker URL: <https://www.docker.com/> (дата обращения 15.02.2021) [1]

⁹ Docker Compose URL: <https://docs.docker.com/compose/> (дата обращения 15.02.2021) [2]

¹⁰ Helm URL: <https://helm.sh/> (дата обращения 15.02.2021) [3]

¹¹ Database As a Service

¹² Страница установки URL:

<https://www.postgresql.org/download/> (дата обращения 15.09.2021) [4]

¹³ Дистрибутив Постгрес для Windows URL:

<https://www.postgresql.org/download/windows/> (дата обращения 15.02.2021) [5]

MacOS X

Ссылка на дистрибутив¹⁴. Опять же варианты установки через GUI¹⁵ или, например, homebrew¹⁶.

Linux

Переходим к самой рекомендованной ОС и самым популярным вариантам установки. Здесь и далее в основном мы будем рассматривать дистрибутив **Ubuntu 20.10 Groovy Gorilla**¹⁷. На других версиях Ubuntu (других дистрибутивах Линукс) все команды будут также работать, хотя возможны незначительные изменения, всё это есть в документации. Вариант GUI мы не рассматриваем, так как в реальных проектах сервера используются без графической оболочки. В варианте работы с GUI устанавливаем или через установку приложений (магазин типа Ubuntu Software или дистрибутив с сайта), или через командную строку и работаем из GUI клиента, например, DBeaver¹⁸.

Остаётся самый используемый вариант - командная строка, и тут есть три варианта:

1. Простая установка - на версиях Ubuntu 20.04 и ниже. Установится дефолтная версия для конкретной ОС. Например, в Ubuntu 20.04 это будет PostgreSQL 12 (*здесь и далее выполняемые нами команды будут помечены полужирным курсивом. Без курсива будет предложен подходящий вариант, но выполнять его не будем*):

`sudo apt install postgresql`

В Ubuntu 20.10 так просто установить Постгрес не получится.

```
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/p/postgresql-12/libpq5_12.5-0ubuntu0.20.10.1_amd64.deb 4
04 Not Found [IP: 35.184.213.5 80]
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/p/postgresql-12/postgresql-client-12_12.5-0ubuntu0.20.10.
1_amd64.deb 404 Not Found [IP: 35.184.213.5 80]
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/p/postgresql-12/postgresql-12_12.5-0ubuntu0.20.10.1_amd64
.deb 404 Not Found [IP: 35.184.213.5 80]
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

Для решения проблемы обновим список пакетов:

`sudo apt update`

И установим сами пакеты:

`sudo apt upgrade -y`

Установим Постгрес:

`sudo apt install postgresql`

¹⁴ Дистрибутив Постгрес под MacOS URL: <https://www.postgresql.org/download/macosx/> (дата обращения 15.02.2021) [6]

¹⁵ GUI (Graphical User Interface) или ГИП (графический интерфейс пользователя)

¹⁶ Homebrew URL: <https://brew.sh/> (дата обращения 15.02.2021) [7]

¹⁷ Ubuntu 20.10 URL: <https://releases.ubuntu.com/groovy/> (дата обращения 15.02.2021) [8]

¹⁸ DBeaver URL: <https://dbeaver.io/> (дата обращения 15.02.2021) [9]

```
aeugene@postgres13:~$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory          Log file
12  main     5432 online postgres /var/lib/postgresql/12/main /var/log/postgresql/postgresql-12-main.log
```

Получилась 12 версия Постгрес (действительно на 23.09.2021). Нам же нужна 13.

Если вы всё-таки установили Постгрес этим способом, его можно удалить:

```
sudo apt remove postgresql
```

2. Расширенная установка, где можно выбрать версию дистрибутива, что конкретно устанавливать (сервер, клиент или всё вместе, а также другие расширения). Рекомендованный вариант. Команды установки приложены на github¹⁹.

Обновим список пакетов и сами пакеты:

```
sudo apt update && sudo apt upgrade -y
```

Добавим репозиторий с последней версией Постгреса:

```
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'
```

Добавим ключ репозитория:

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
```

Обновим список пакетов и установим Постгрес:

```
sudo apt-get update && sudo apt-get -y install postgresql
```

Для удобства все вышеперечисленные команды соберём в одну, используя символы **&&**:

```
aeugene@postgres13:~$ sudo apt update && sudo apt upgrade -y && sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list' && wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add - && sudo apt-get update && sudo apt-get -y install postgresql
```

```
Success. You can now start the database server using:
```

```
pg_ctlcluster 13 main start
```

```
Ver Cluster Port Status Owner    Data directory          Log file
13  main     5432 down   postgres /var/lib/postgresql/13/main /var/log/postgresql/postgresql-13-main.log
```

Кластер Постгреса успешно установлен и настроен по умолчанию. Также по умолчанию при установке кластер автоматически запускается. Что значат конкретные значения 13, main и т.д. мы разберём в следующей главе “Физический уровень”.

¹⁹ Установка PostgreSQL 13 URL:

https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/01/install.txt (дата обращения 07.03.2021) [10]

3. Сборка PostgreSQL из пакетов (вариант для гурманов). В основном это предназначено для применения кастомных патчей - они модифицируют исходный код на С для внесения изменений в ядро СУБД. Позволяет значительно расширить функционал или кастомизировать поведение. Единственный минус этого варианта, кроме сложности развертывания, это обновление версии Постгреса - так как исходники, естественно, меняются от версии к версии и нужно заново изменять патч, а если это патч от сторонней организации...

Вот один из примеров кастомного патча, с помощью которого можно закрыть изменение данных в таблице напрямую даже от суперпользователя БД²⁰ - <https://github.com/AbdulYadi/postgresql-private>²¹.

Для **сборки PostgreSQL из пакетов** мы должны проделать следующие шаги:

Скачиваем нужную версию Постгреса с FTP²². В данном случае нам нужна версия 12.1²³:

```
wget https://ftp.postgresql.org/pub/source/v12.1/postgresql-12.1.tar.gz
```

Скачаем патч из репозитория:

```
wget https://github.com/AbdulYadi/postgresql-private
```

Применим патч:

```
run patch -p0 < path-to-downloaded-patch-file
```

Соберём Постгрес из исходников, следуя командам из файла INSTALL в каталоге Постгреса:

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

Как видим, всё не так просто.

²⁰ База данных

²¹ Github AbdulYadi URL: <https://github.com/AbdulYadi/postgresql-private> (дата обращения 15.02.2021) [11]

²² File Transfer Protocol URL: https://en.wikipedia.org/wiki/File_Transfer_Protocol (дата обращения 07.03.2021) [12]

²³ PostgreSQL 12.1 URL: <https://ftp.postgresql.org/pub/source/v12.1/postgresql-12.1.tar.gz> (дата обращения 07.03.2021) [13]

Теперь при попытке модифицировать табличку с приватным модификатором получим следующую ошибку:

INSERT INTO public.test VALUES (1, 'abc');

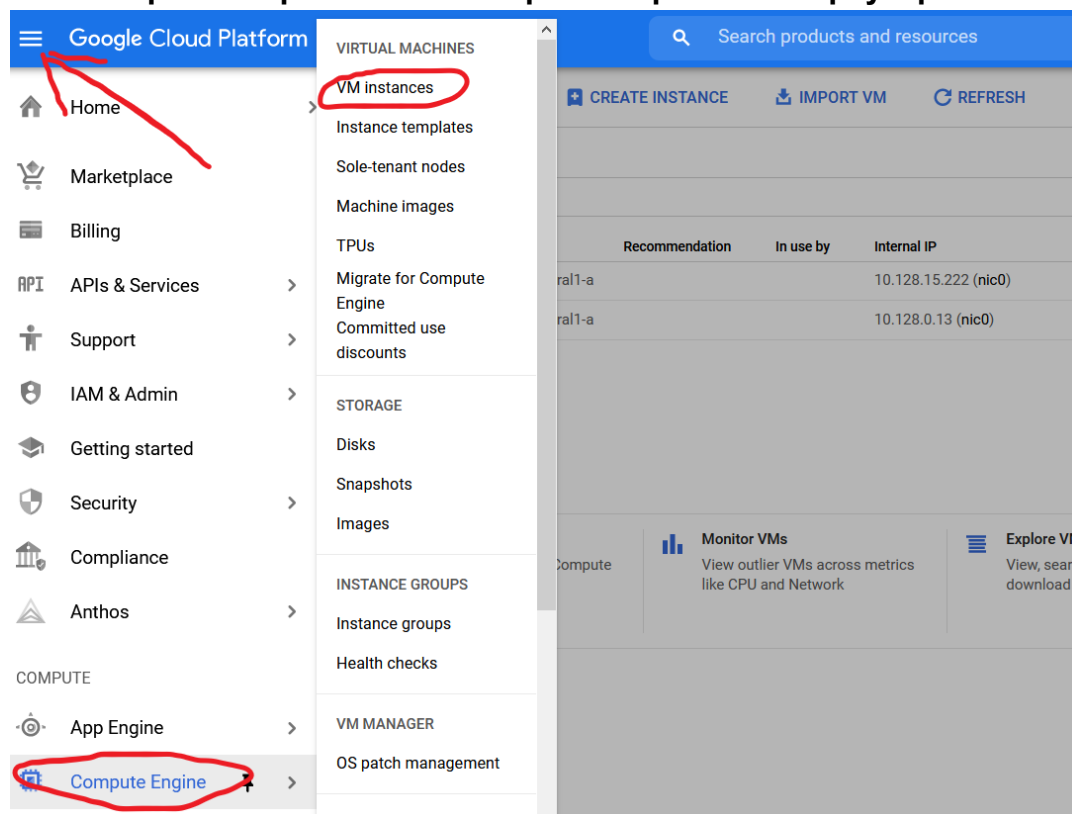
ERROR: do not modify table with "private modify" option outside SQL, PLPGSQL or other SPI-based function

Модифицировать данные сможем только через хранимые функции с нужными правами.

Все предыдущие варианты установки на ОС Linux можно выполнить как на отдельной физической машине, так и на виртуальной. Сейчас и в дальнейшем мы будем использовать VM²⁴ в облаке Google²⁵. Варианты в облаках AWS, Azure и Яндекс Облако будут в дальнейших главах.

VM будем разворачивать в пространстве GCE²⁶. Есть **два простых варианта**, как развернуть VM:

Первый вариант - можно прямо через GUI в браузере.



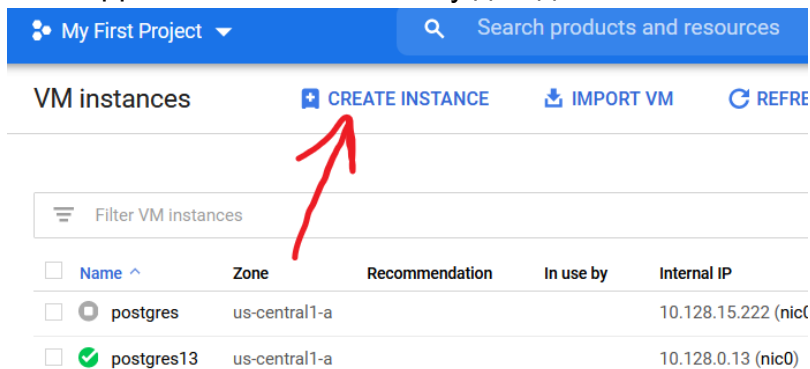
В левом верхнем углу нажимаем на кнопку выбора продукта, затем выбираем Compute Engine и VM Instances.

²⁴ Виртуальная машина

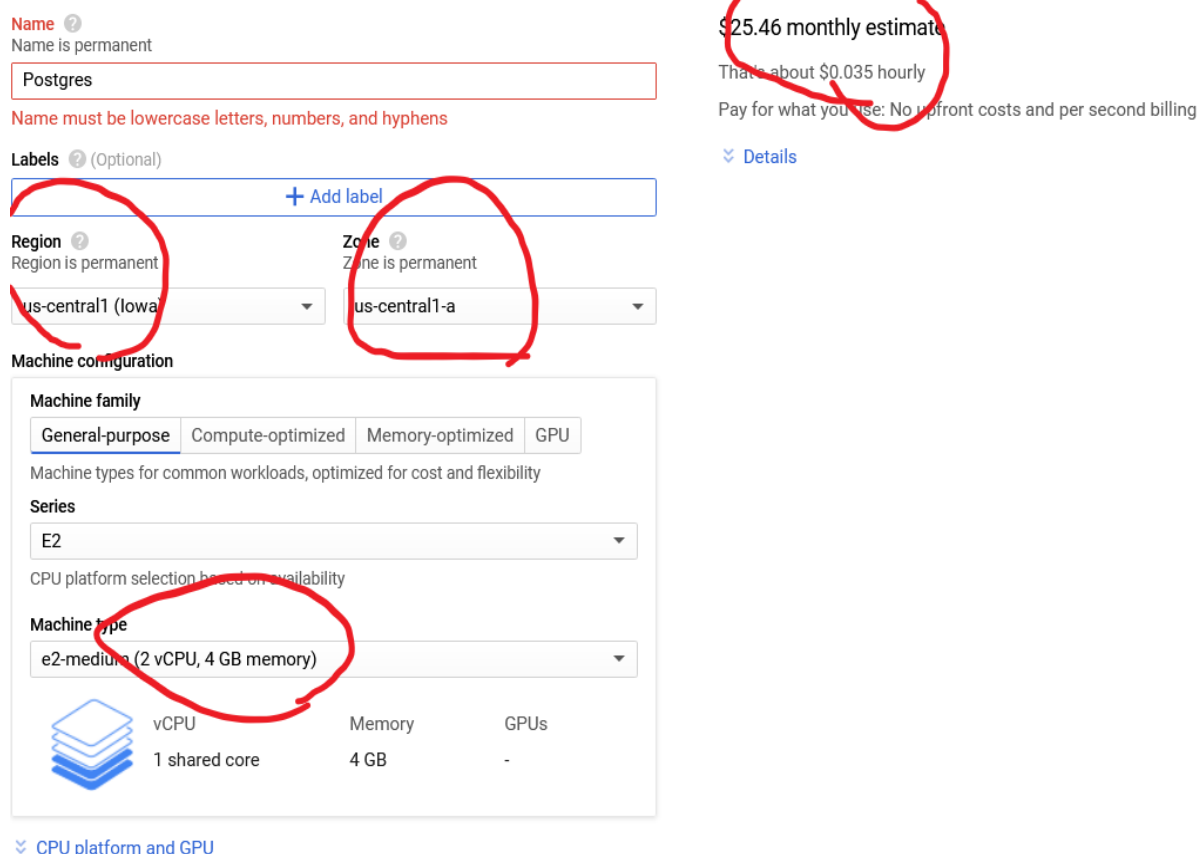
²⁵ Google Cloud Platform URL: <https://cloud.google.com/> (дата обращения 07.03.2021) [14]

²⁶ Google Compute Engine

Далее нажимаем кнопку для добавления ВМ - Create Instance.



После этого выбираем тип ВМ и её характеристики. Чем мощнее ВМ, тем больше в итоге мы заплатим. Оплата начисляется в зависимости от времени работы ВМ, так что общая рекомендация - выключать ВМ, когда не используете.



Несколько замечаний: имя инстанса может состоять только из латинских букв в нижнем регистре, цифр и дефисов. Оно должно быть уникально в рамках вашего проекта.

Здесь же можем выбрать регион, а в нём зону, где будет физически расположена ВМ - в зависимости от этого будет рассчитана стоимость владения. Чем ближе к вам физически будет ВМ, тем меньше пинг²⁷ и выше скорость передачи.


²⁷ Ping URL: <https://en.wikipedia.org/wiki/Ping> (дата обращения 07.03.2021) [15]

Также можно выбрать количество виртуальных ядер и памяти. По умолчанию, типа VM e2-medium (2 ядра и 4 гигабайта памяти) нам хватит для небольшого инстанса Постгреса.

Справа вверху будет указана стоимость за месяц, если VM будет включена 24/7. Если VM выключена, с нас будут списываться деньги только за использование диска.

Далее выбираем ОС, размер и тип жёсткого диска нашей VM.

Boot disk ?

 New 10 GB balanced persistent disk
Image
Debian GNU/Linux 10 (buster) Change

Operating system
Ubuntu

Version
Ubuntu 20.10


amd64 groovy image built on 2021-02-24, supports Shielded VM features ?

Boot disk type ? Size (GB) ?

SSD persistent disk 10

В данном случае мы выбрали Ubuntu Groovy и SSD²⁸-диск на 10 Гб. Соответственно, чем больше и быстрее выбираем диск, тем больше будет стоимость месячного владения. После выбора можем проверить, насколько увеличилась цифра оплаты за месяц. Остальные настройки VM мы будем рассматривать в дальнейших главах по мере необходимости.

Второй вариант: в самом низу, рядом с кнопкой создания VM, есть кнопка **command line**, нажав которую получим код для выполнения **утилитой cli²⁹ gcloud³⁰**.

You will be billed for this instance. [Compute Engine pricing](#) 

Create Cancel

Equivalent [REST](#) or [command line](#)

²⁸ SSD URL: https://en.wikipedia.org/wiki/Solid-state_drive (дата обращения 07.03.2021) [16]

²⁹ Command line interface

³⁰ Gcloud URL: <https://cloud.google.com/sdk/gcloud> (дата обращения 07.03.2021) [17]

gcloud command line

This is the gcloud command line with the parameters you have selected.

```
gcloud beta compute --project=celtic-house-266612 instances create ostgres-2 --zone=us-central1-a --machine-type=e2-small --subnet=default --network-tier=PREMIUM --maintenance-policy=MIGRATE --service-account=933982307116-compute@developer.gserviceaccount.com --scopes=https://www.googleapis.com/auth/devstorage.read_only,https://www.googleapis.com/auth/logging.write,https://www.googleapis.com/auth/monitoring.write,https://www.googleapis.com/auth/servicecontrol,https://www.googleapis.com/auth/service.management.readonly,https://www.googleapis.com/auth/trace.append --image=debian-10-buster-v20210217 --image-project=debian-cloud --boot-disk-size=10GB --boot-disk-type=pd-balanced --boot-disk-device-name=ostgres-2 --no-shielded-secure-boot --shielded-vtpm --shielded-integrity-monitoring --reservation-affinity=any
```

☒ Line wrapping

[CLOSE](#) [RUN IN CLOUD SHELL](#)

И, опять же, есть два варианта:

- нажать **run in cloud shell** - прямо в браузере откроется эмуляция терминала, можно оттуда выполнить команду развёртывания ВМ и в дальнейшем работать через браузерный терминал
- **скопировать команду** и использовать собственный терминал (ак мы и будем использовать **cli gcloud** в дальнейшем)

В нашем случае развернём ВМ с 2 ядрами и 4 Гб памяти на ОС Ubuntu Groovy с жёстким диском SSD на 10 Гб:

```
aeugene@Aeuge:/mnt/c/Users/arist$ gcloud beta compute --project=celtic-house-266612 instances create postgres13 --zone=us-central1-a --machine-type=e2-medium --subnet=default --network-tier=PREMIUM --maintenance-policy=MIGRATE --service-account=933982307116-compute@developer.gserviceaccount.com --scopes=https://www.googleapis.com/auth/devstorage.read_only,https://www.googleapis.com/auth/logging.write,https://www.googleapis.com/auth/monitoring.write,https://www.googleapis.com/auth/servicecontrol,https://www.googleapis.com/auth/service.management.readonly,https://www.googleapis.com/auth/trace.append --image=ubuntu-2010-groovy-v20210211a --image-project=ubuntu-os-cloud --boot-disk-size=10GB --boot-disk-type=pd-ssd --boot-disk-device-name=postgres13 --no-shielded-secure-boot --shielded-vtpm --shielded-integrity-monitoring --reservation-affinity=any

Created [https://www.googleapis.com/compute/beta/projects/celtic-house-266612/zones/us-central1-a/instances/postgres13].
WARNING: Some requests generated warnings:
- The resource 'projects/ubuntu-os-cloud/global/images/ubuntu-2010-groovy-v20210211a' is deprecated. A suggested replacement is 'projects/ubuntu-os-cloud/global/images/ubuntu-2010-groovy-v20210224'.

NAME          ZONE          MACHINE_TYPE  PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP    STATUS
postgres13    us-central1-a  e2-medium     false        10.128.0.13   34.123.177.234  RUNNING
```

Через несколько секунд получим ВМ с двумя IP-адресами³¹: внутренним, доступным только в кластере 10.128.* и внешним, доступным из интернета 34.123.177.234. По умолчанию к только что развёрнутому Постгресу по сети доступа не имеем, а как открыть доступ обсудим в следующей главе.

³¹ IP URL: https://en.wikipedia.org/wiki/IP_address (дата обращения 07.03.2021) [18]

Соответственно, чтобы получить **доступ к ВМ**, есть три пути:

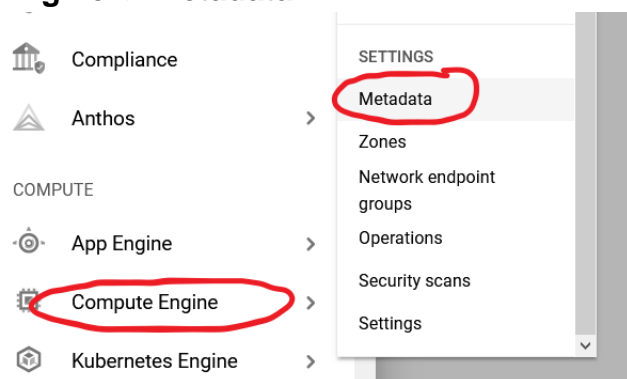
- через браузер и виртуальный терминал
- используя клиент **ssh**³² - доступ по защищённому шифрованному протоколу
- используя утилиту **gcloud + ssh**

Настройка ssh.

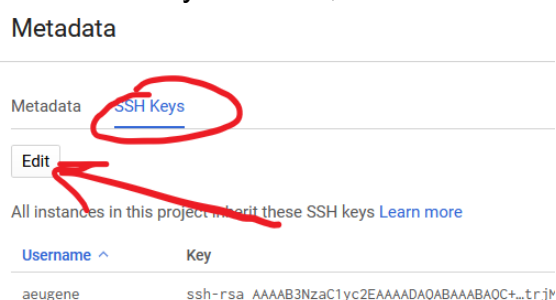
В данном случае настраиваем в Linux. Для Windows и MacOS есть свои ssh-клиенты и настройка примерно такая же:

- на своей машине создаём каталог: **mkdir \$HOME\ssh**
- даём права на него: **chmod 700 \$HOME\ssh**
- генерируем ssh-ключ: **ssh-keygen имя**
- добавляем его: **ssh-add имя**
- выводим на экран и копируем ключ: **cat имя.pub**
- добавляем свой ssh-ключ в GCE metadata

Для добавления ssh-ключа в GCE необходимо выбирать пункты **Compute Engine -> Metadata**.



Далее вкладка SSH keys, кнопка edit, добавляем свой public ssh-ключ, нажав кнопку add item, и вставляем из буфера обмена в окно наш ключ.



³² SSH URL: [https://en.wikipedia.org/wiki/SSH_\(Secure_Shell\)](https://en.wikipedia.org/wiki/SSH_(Secure_Shell)) (дата обращения 07.03.2021) [19]

Enter public SSH key

✕

+ Add item

Save

Cancel

Теперь зайти со своего хоста очень просто:

ssh имя@ip созданной ВМ

В дальнейших главах будем использовать **вариант** подключения по **ssh**, используя утилиту **gcloud** и имя инстанса:

gcloud compute ssh postgres13

```
aeugene@Aeuge:/mnt/c/Users/arist$ gcloud compute ssh postgres13
No zone specified. Using zone [us-central1-a] for instance: [postgres13].
Warning: Permanently added 'compute.5577661446816869047' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 20.10 (GNU/Linux 5.8.0-1022-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Mar  7 08:54:14 UTC 2021

System load:  0.25          Processes:            106
Usage of /:   15.0% of 9.52GB Users logged in:          0
Memory usage: 5%           IPv4 address for ens4: 10.128.0.13
Swap usage:   0%

aeugene@postgres13:~$
```

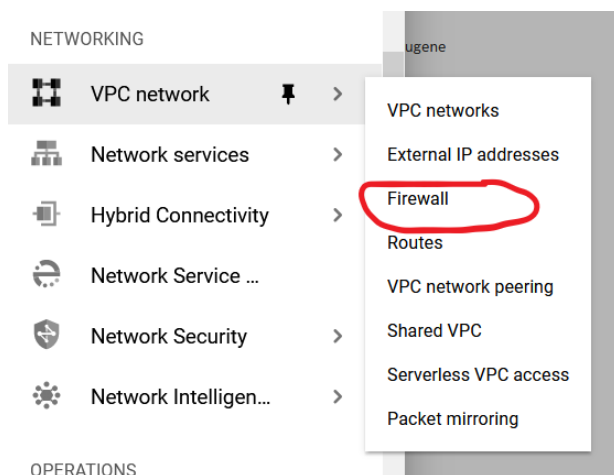
Далее просто устанавливаем Постгрес 13, как было описано в главе ранее:

```
sudo apt update && sudo apt upgrade -y && sudo sh -c 'echo "deb
http://apt.postgresql.org/pub/repos/apt $(lsb_release
-cs)-pgdg main" >
/etc/apt/sources.list.d/pgdg.list' && wget --quiet -O -
https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add - && sudo apt-get
update && sudo apt-get -y install postgresql
```

После установки Постгреса нам также будет необходимо настроить фаервол в GCP для доступа извне по порту 5432.

Настройка фаервола в Google.

Выбираем пункт **VPC network -> Firewall**.



Далее нажимаем **Create Firewall Rule** и, после задания имени правилу, прописываем, с каких адресов разрешено подключение, для каких экземпляров ВМ и по каким портам³³.

← Create a firewall rule

☐ Deny

Targets
Specified target tags

Target tags *

At least one tag is required

Source filter
IP ranges

Source IP ranges *

Second source filter
None

Protocols and ports ?

☐ Allow all

☒ Specified protocols and ports

☒ tcp : 5432

☐ udp : all

Для начала, нужно или ввести тег машин, для которых открываем доступ, или выбрать пункт - все инстансы (в нашем случае).

Порт Постгреса по умолчанию 5432.

Также вводим маску подсети, откуда разрешён доступ: 0.0.0.0/0.

Нажимаем кнопку “Сохранить”, и через несколько минут доступ извне будет открыт, но при этом у Постгреса есть ещё встроенный фаервол.

³³ Port URL: [https://en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking)) (дата обращения 07.03.2021) [20]

Как настроить доступ изнутри Постгреса узнаем в следующей главе.

Предложенные настройки для широкого доступа не рекомендованы на продакшн-системах, мы их используем только для простоты демонстрации наших тестов.

Открытие портов Постгреса в интернет я не рекомендую, максимум, во внутреннюю сеть вашего гугл-проекта.

Использование PostgreSQL 13 в докере.

Здесь всё просто. Установим докер на машину по инструкции³⁴. Создаём контейнер с 13 Постгресом, указываем пароль, открываем порт 5432 для доступа в контейнер и указываем, куда внутрь контейнера примонтировать локальную директорию:

```
sudo docker run --name pg-server -e POSTGRES_PASSWORD=postgres -d -p 5432:5432 -v /var/lib/postgres:/var/lib/postgresql/data postgres:13
```

Проверим подключение используя встроенную утилиту для работы с Постгресом **psql**, укажем хост и порт для подключения. Ключ **-W** нужен для указания пароля:

```
psql -h localhost -U postgres -W
```

```
aeugene@Aeuge:/mnt/c/Users/arist$ sudo docker run --name pg-server -e POSTGRES_PASSWORD=postgres -d -p 5432:5432 -v /var/lib/postgres:/var/lib/postgresql/data postgres:13
6d49d27d7d56da46e43fd8fe0736d9fa6b3d07385895980148551d81efe61479
aeugene@Aeuge:/mnt/c/Users/arist$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
6d49d27d7d56   postgres:13    "docker-entrypoint.s..." 11 seconds ago Up 8 seconds  0.0.0.0:5432->5432/tcp
aeugene@Aeuge:/mnt/c/Users/arist$ psql -h localhost -U postgres -W
Password:
psql (12.4 (Ubuntu 12.4-1.pgdg18.04+1), server 13.1 (Debian 13.1-1.pgdg100+1))
WARNING: psql major version 12, server major version 13.
         Some psql features might not work.
Type "help" for help.

postgres=# |
```

В данном случае мы запустили контейнер с 13 Постгресом и подключились к нему с localhost под пользователем postgres.

Не забываем сопоставить директорию с данными БД в контейнере и локальный каталог, так как все внутренние данные в контейнере будут удалены после рестарта контейнера.

³⁴ Установка docker URL: <https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04-ru> (дата обращения 23.09.2021) [21]

Использовать **docker compose** для создания кластера Постгреса тоже просто. Устанавливаем docker compose по инструкции³⁵. Создаём файл **docker-compose.yml** со следующим содержанием³⁶:

```
version: '3.1'
volumes:
  pg_project:
services:
  pg_db:
    image: postgres:13
    restart: always
    environment:
      - POSTGRES_PASSWORD=secret
      - POSTGRES_USER=postgres
      - POSTGRES_DB=stage
    volumes:
      - pg_project:/var/lib/postgresql/data
    ports:
      - ${POSTGRES_PORT:-5432}:5432
```

Поднимаем Постгрес используя docker compose:

docker-compose up -d

- ключ **up** - запустить конфигурацию, описанную в файле docker-compose.yml в текущей директории
- ключ **-d** - запустить в качестве демона (фоновый процесс), а не интерактивно

Обратите внимание на указание версии контейнера с Постгресом:

image: postgres:13

Рекомендую всегда (!!!) указывать версию контейнера, чтобы не было сюрпризов при обновлении версии.

³⁵ Установка docker compose URL: <https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04-ru> (дата обращения 23.09.2021) [22]

³⁶ docker-compose.yml URL: https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/01/docker-compose.yml (дата обращения 07.03.2021) [23]

Проверим подключение:

psql -h localhost -U postgres -W

```
aeugene@Aeuge:/mnt/c/Users/arist$ sudo nano docker-compose.yml
aeugene@Aeuge:/mnt/c/Users/arist$ docker-compose up -d
Creating network "arist_default" with the default driver
Creating volume "arist_pg_project" with default driver
Creating arist_pg_db_1 ... done
aeugene@Aeuge:/mnt/c/Users/arist$ psql -h localhost -U postgres -W
Password:
psql (12.4 (Ubuntu 12.4-1.pgdg18.04+1), server 13.1 (Debian 13.1-1.pgdg100+1))
WARNING: psql major version 12, server major version 13.
        Some psql features might not work.
Type "help" for help.

postgres=#
```

Чтобы остановить docker compose, выполним:

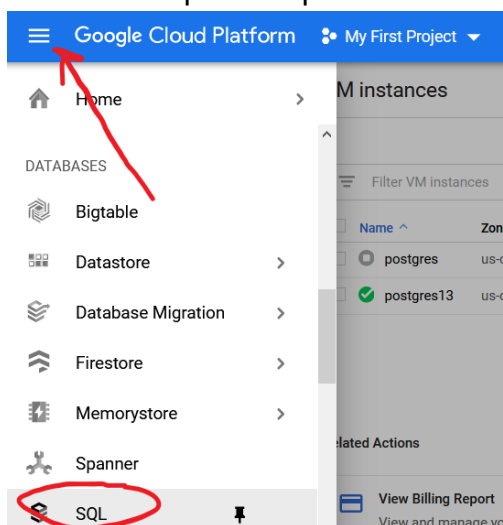
docker-compose down

Про использование **kubernetes** для развёртывания Постгреса у нас будет целая глава далее, так что здесь это рассматривать не будем.

Остаётся вариант **использования Постгреса как DBaaS-решения**. Тема использования БД как сервиса всё больше приобретает популярность. Теперь нам не надо поддерживать ОС, версию Постгреса и другие настройки, нужно только настроить доступ по порту. От нас требуется только схема данных и сами данные. Можем на ходу добавить мощности для БД, увеличить место для хранения или создать реплики для чтения. Более подробно рассмотрим настройки Cloud SQL - версия DBaaS от Google - в соответствующей главе.

Сейчас настроим самый простой инстанс Постгреса как сервиса и протестируем к нему доступ.

Выбираем сервис Cloud SQL.



Далее выбираем **Create Instance**.

MySQL Versions: 5.6, 5.7, 8.0 → Choose MySQL	PostgreSQL Versions: 9.6, 10, 11, 12, 13 → Choose PostgreSQL	SQL Server Versions: 2017 → Choose SQL Server
---	---	--

Выбираем 13 Постгрес.

← Create a PostgreSQL instance

Instance info

Instance ID *

postgres

Use lowercase letters, numbers, and hyphens. Start with a letter.

Password *

••••••

GENERATE

Database version *

PostgreSQL 13

Choose region and zonal availability

For better performance, keep your data close to the services that need it. Region is permanent, while zone can be changed any time.

Region

us-central1 (Iowa)

Single zone

In case of outage, no failover. Not recommended for production.

Multiple zones (Highly available)

Automatic failover to another zone within your selected region. Recommended for production instances. Increases cost.

Summary

Region	us-central1 (Iowa)
DB Version	PostgreSQL 13
vCPUs	4 vCPU
Memory	26 GB
Storage	100 GB
Network throughput (MB/s)	1,000 of 2,000
Disk throughput (MB/s)	Read: 48.0 of 240.0 Write: 48.0 of 240.0
IOPS	Read: 3,000 of 15,000 Write: 3,000 of 15,000
Connections	Public IP
Backup	Automated
Availability	Multiple zones (Highly available)
Point-in-time recovery	Enabled

Обратите внимание на следующие моменты:

- выбирайте регион, где будет расположен инстанс, на котором будет расположен ваш Постгрес исходя из следующих вопросов:
 - расстояние до региона - чем ближе, тем быстрее
 - стоимость инстанса в данном регионе
- внутри это тоже ВМ, только у вас не будет туда доступа, и управляться она будет автоматически
- обратите внимание на включённую по умолчанию возможность мультизоны для высокой доступности (high availability). Это автоматически увеличивает стоимость владения в два раза минимум. Рекомендовано для продакшн систем, обеспечивает отказоустойчивость при смерти primary инстанса и автоматически переключает на secondary. Но при этом наша запасная нода не участвует в обработке запросов. Более тонкая настройка будет показана в соответствующей главе

Итого, если обратить внимание на обведённый круг справа, имеем инстанс со следующими характеристиками:

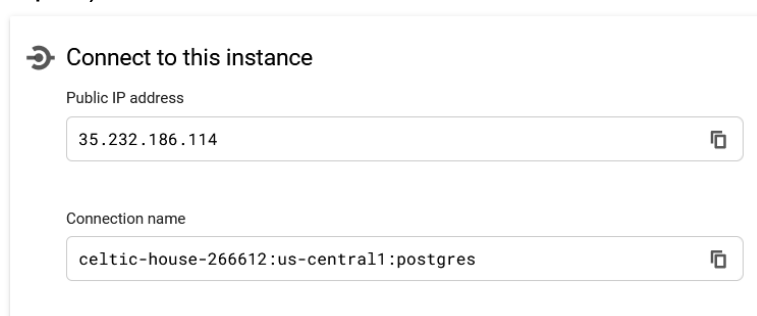
- 4 ядра
- 26 Гб памяти
- 100 Гб диск
- гарантированные минимальные скорости сети/диска

- внешний IP-адрес
- автоматический бэкап (не бесплатно)
- мультизональную доступность (не бесплатно)
- возможность восстановления во времени PITR³⁷

Сколько же это стоит, спросите вы?

А вот сюрприз - узнаем уже после создания, посмотрев финансовый отчёт!!!

После создания инстанса мы получаем внешний IP. Но чтобы получить к нему доступ, нужно настроить брандмауэр (он же фаервол, он же межсетевой экран)³⁸:



➔ Connect to this instance

Public IP address

35.232.186.114

Connection name

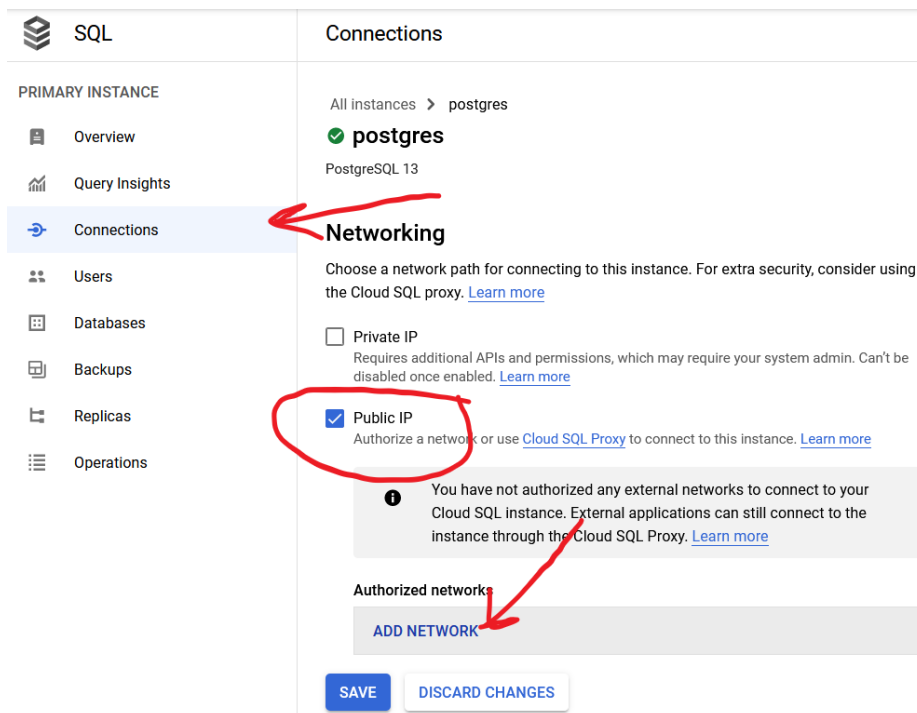
celtic-house-266612:us-central1:postgres

Есть ещё вариант подключения используя **Cloud Proxy**, но его мы будем рассматривать в главе про Google Cloud.

Настроим фаервол.

³⁷ Point in time recovery - восстановление на определенную точку времени


³⁸ Firewall URL: [https://en.wikipedia.org/wiki/Firewall_\(computing\)](https://en.wikipedia.org/wiki/Firewall_(computing)) (дата обращения 07.03.2021) [24]



Выберем Connections в меню слева. И видим, что, несмотря на галочку Public IP, публичного доступа у нас нет. Нам **нужно добавить IP-адрес/подсеть**³⁹, с которой нам будет разрешён доступ.

Нажмём кнопку Add Network.

☒ Public IP
Authorize a network or use [Cloud SQL Proxy](#) to connect to this instance. [Learn more](#)

 You have added 0.0.0.0/0 as an allowed network. This prefix will allow any IPv4 client to pass the network firewall and make login attempts to your instance, including clients you did not intend to allow. Clients still need valid credentials to successfully log in to your instance.

Authorized networks

New network

Name
postgres

Use [CIDR notation](#)

Network *
0.0.0.0/0
Example: 199.27.25.0/24

CANCEL DONE

В данном случае мы разрешили доступ с любого ip 0.0.0.0/0. Повторяю, что так делать на рабочих системах однозначно не стоит. Да и вообще доступ из

³⁹ Маска подсети URL: <https://en.wikipedia.org/wiki/Subnetwork> (дата обращения 07.03.2021) [25]

интернета к БД не рекомендован. Используйте вместо этого доступ из интернета через backend⁴⁰ по REST⁴¹.

После нажатия кнопок **Done** и **Save**, через пару минут доступ по IP окажется открыт. Давайте проверим:

`psql -h 35.232.186.114 -U postgres -W`

```
aeugene@Aeuge:/mnt/c/Users/arist$ psql -h 35.232.186.114 -U postgres -W
Password:
psql (12.4 (Ubuntu 12.4-1.pgdg18.04+1), server 13.1)
WARNING: psql major version 12, server major version 13.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=> |
```

Таким образом, мы с вами рассмотрели основные варианты развёртывания PostgreSQL. Но у нас остались ещё две затронутые темы по настройке доступа, они будут рассмотрены в следующих главах:

- **настройка ssh** - для обеспечения защищённого подключения
- **настройка firewall в Постгресе** - для доступа к инстансу из интернета

Итого, в этой главе мы рассмотрели различные варианты установки Постгреса и в самом конце настроили доступ к нашему инстансу извне, прописав ssh-ключ и создав правило в фаерволе. В следующей главе мы рассмотрим, как открыть доступ к БД изнутри Постгреса.

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/01/install.txt

⁴⁰ Backend URL: https://en.wikipedia.org/wiki/Front_end_and_back_end (дата обращения 07.03.2021) [26]

⁴¹ REST URL: https://en.wikipedia.org/wiki/Representational_state_transfer (дата обращения 07.03.2021) [27]

2. Физический уровень

В предыдущей главе мы рассмотрели варианты установки 13 Постгреса. Далее будем рассматривать вариант установки Постгреса на ВМ под ОС Ubuntu Groovy.

Все команды из этой главы доступны в файле на гитхабе https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/02/physical_level.txt. Для каждой последующей главы есть свой соответствующий файл.

После установки Постгреса доступ к нему открыт только с локальной машины и только из-под рута⁴². Через утилиту `gcloud` посмотрим список ВМ и заодно узнаем внешний адрес для подключения:

```
aeugene@Aeuge:/mnt/c/Users/arist$ gcloud compute instances list
```

NAME	ZONE	MACHINE_TYPE	PREEMPTIBLE	INTERNAL_IP	EXTERNAL_IP	STATUS
postgres	us-central1-a	e2-medium		10.128.15.222		TERMINATED
postgres13	us-central1-a	e2-medium		10.128.0.13	34.66.131.159	RUNNING

Можно напрямую подключиться к ВМ через `ssh`, но мы будем использовать здесь и в дальнейшем команду:

`gcloud compute ssh postgres13`

После подключения посмотрим на список кластеров Постгреса через утилиту `pg_lsclusters` (в Постгресе принята такая терминология, что каждый инстанс называется кластером):

`pg_lsclusters`

```
aeugene@postgres13:~$ pg_lsclusters
```

Ver	Cluster	Port	Status	Owner	Data directory	Log file
13	main	5432	online	postgres	/var/lib/postgresql/13/main	/var/log/postgresql/postgresql-13-main.log

Видим, что у нас пока развёрнут один кластер **13** версии **Постгреса** по имени **main** на порту **5432**.

Кластер находится онлайн.

Владелец кластера линукс-пользователь **postgres**.

*Обратите внимание, что, несмотря на идентичное имя **postgres** у суперпользователя Постгреса, это другой тип - пользователь ОС Линукс.*

Также видим, в каком каталоге находятся файлы с базой данных и файлы с логами. Параметры **cluster**, **port**, **data directory** и **log file** должны быть уникальны для одной ВМ.

⁴² Пользователь `root` - суперпользователь Линукс с максимальными правами.

На самом деле утилит Постгреса **pg_*** в Ubuntu много. Их список можно получить, используя функцию автодописывания команды, набрав **pg_** и нажав кнопку **tab** два раза:

```
aeugene@postgres13:~$ pg_
pg_archivecleanup  pg_conftool      pg_dump          pg_receivewal    pg_restore
pg_basebackup      pg_createcluster  pg_dumpall       pg_receivexlog   pg_updatedb
pg_builtex          pg_ctlcluster     pg_isready       pg_recvlogical   pg_upgradecluster
pg_config           pg_dropcluster    pg_lsclusters    pg_renamecluster pg_virtualenv
```

Рассмотрим основные из них:

- **pg_createcluster**⁴³ - создать кластер с нужными параметрами
- **pg_renamecluster**⁴⁴ - переименовать кластер
- **pg_dropcluster**⁴⁵ - удалить кластер
- **pg_ctlcluster**⁴⁶ - утилита для запуска и остановки кластера
- **pg_config** - утилита изменения параметров кластера без ручной правки конфиг файлов
- **pg_basebackup**, **pg_dump**, **pg_dumpall**, **pg_receivewal**, **pg_restore** будут рассмотрены в главах по бэкапам и репликации

Например, добавим ещё один кластер:

sudo pg_createcluster 13 main2

```
aeugene@postgres13:~$ pg_createcluster 13 main2
install: cannot change permissions of '/etc/postgresql/13/main2': No such file or directory
Error: could not create configuration directory; you might need to run this program with root privileges
aeugene@postgres13:~$ sudo pg_createcluster 13 main2
Creating new PostgreSQL cluster 13/main2 ...
/usr/lib/postgresql/13/bin/initdb -D /var/lib/postgresql/13/main2 --auth-local peer --auth-host md5
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 "C.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/13/main2 ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Etc/UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

Success. You can now start the database server using:

    pg_ctlcluster 13 main2 start

Ver Cluster Port Status Owner    Data directory                   Log file
13 main2    5433 down   postgres /var/lib/postgresql/13/main2 /var/log/postgresql/postgresql-13-main2.log
aeugene@postgres13:~$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory                   Log file
13 main    5432 online postgres /var/lib/postgresql/13/main /var/log/postgresql/postgresql-13-main.log
```

⁴³ **pg_createcluster** URL: https://helpmanual.io/help/pg_createcluster/ (дата обращения 10.03.2021) [1]

⁴⁴ **pg_renamecluster** URL: <https://www.onworks.net/programs/pg-renamecluster-online> (дата обращения 10.03.2021) [2]

⁴⁵ **pg_dropcluster** URL: http://manpages.ubuntu.com/manpages/trusty/man8/pg_dropcluster.8.html (дата обращения 10.03.2021) [3]

⁴⁶ **pg_ctlcluster** URL: http://manpages.ubuntu.com/manpages/hirsute/en/man1/pg_ctlcluster.1.html (дата обращения 10.03.2021) [4]

Обратите внимание на следующие особенности:

- под нашим простым линукс-пользователем кластер создать не удалось, так как нет доступа к каталогу **/etc/postgresql/13**. Тут было три варианта:
 - переключиться на линукс-пользователя postgres командой **sudo su postgres** и из-под него создать кластер
 - перейти под линукс-суперпользователя **root** командой **sudo su**
 - или, как я сделал, использовать утилиту **sudo** для поднятия своих прав до прав суперпользователя
- кластер создался, но находится в остановленном состоянии

Важно отметить, что именно на этом этапе задаются дефолтные настройки для кодировок и правил сортировки - локали. В данном случае это UTF8 и C.UTF-8. **Порт 5433 задался автоматически**. Если нам нужен другой порт - нужно было указать при создании кластера. Теперь для его изменения нужно остановить кластер и поменять параметр **port** в файле **postgresql.conf**. Например, используя редактор **nano**, следующей командой: **sudo nano /etc/postgresql/13/main/postgresql.conf**. После внесения изменений нажмите **Control+X** и подтвердите изменения в файле, нажав кнопки **Y** и потом **ENTER**. Запустите кластер - кластер будет уже запущен на новой порту.

Посмотрим на физическую структуру каталога с данными в Постгресе. Для этого **зайдём под линукс-пользователем postgres**:

sudo su postgres

Перейдём в этот каталог и выполним команду:

ls -la

```
aeugene@postgres13:~$ sudo su postgres
postgres@postgres13:/home/aeugene$ cd /var/lib/postgresql/13/main
postgres@postgres13:~/13/main$ ls -la
total 92
drwx----- 19 postgres postgres 4096 Mar 10 11:32 .
drwxr-xr-x  4 postgres postgres 4096 Mar 10 12:18 ..
-rw-----  1 postgres postgres   3 Mar  7 09:24 PG_VERSION
drwx-----  5 postgres postgres 4096 Mar  7 09:24 base
drwx-----  2 postgres postgres 4096 Mar 10 11:32 global
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_commit_ts
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_dynshmem
drwx-----  4 postgres postgres 4096 Mar 10 11:37 pg_logical
drwx-----  4 postgres postgres 4096 Mar  7 09:24 pg_multixact
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_notify
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_replslot
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_serial
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_snapshots
drwx-----  2 postgres postgres 4096 Mar 10 11:32 pg_stat
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_stat_tmp
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_subtrans
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_tblspc
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_twophase
drwx-----  3 postgres postgres 4096 Mar  7 09:24 pg_wal
drwx-----  2 postgres postgres 4096 Mar  7 09:24 pg_xact
-rw-----  1 postgres postgres   88 Mar  7 09:24 postgresql.auto.conf
-rw-----  1 postgres postgres  130 Mar 10 11:32 postmaster.opts
-rw-----  1 postgres postgres  107 Mar 10 11:32 postmaster.pid
```

Здесь нас будет интересовать файл **postgresql.auto.conf**, в котором лежат настройки кластера, которые перезаписывают дефолтные значения. То есть мы во время работы БД изменяем какой-нибудь системный параметр - это значение попадает в этот файл и будет применено в последнюю очередь при рестарте инстанса, и перезапишет значение из файла **postgresql.conf**. Подробнее про эти файлы поговорим в главе про настройки Постгреса.

В каталоге **global** находятся глобальные/системные объекты, которые присутствуют во всех БД кластера.

В каталоге **base** как раз расположены файлы с нашими базами данных. Но используются не имена баз данных, а их **OID - object id**:

ls -l base

```
postgres@postgres13:~/13/main$ ls -l base
total 12
drwx----- 2 postgres postgres 4096 Mar 10 11:32 1
drwx----- 2 postgres postgres 4096 Mar  7 09:24 13444
drwx----- 2 postgres postgres 4096 Mar 10 12:54 13445
```

Узнать, какой номер какой БД соответствует, можно, выполнив запрос в **psql**:

SELECT oid, datname, dattablespace FROM pg_database;

```
postgres=# select oid, datname, dattablespace from pg_database;
 oid | datname | dattablespace
-----+-----+-----
 13445 | postgres |          1663
    1 | template1 |          1663
 13444 | template0 |          1663
(3 rows)
```

Здесь видим кроме непонятных баз данных ещё и такую штуку как **dattablespace** - табличное пространство. Давайте разбираться в этом по порядку.

PostgreSQL кластер это несколько баз данных под управлением одного сервера. По умолчанию присутствуют:

- template0
- template1
- postgres

template0 используется:

- для восстановления из резервной копии
- по умолчанию даже нет прав на connect
- не рекомендовано вносить изменения - лучше всего не создавать в ней никаких объектов

template1 используется:

- как шаблон для создания новых баз данных
- в нём имеет смысл делать некие действия, которые не хочется делать каждый раз при создании новых баз данных - создать

хранимые функции/процедуры, создать справочники, включить расширения и т.п.

postgres используется:

- первая база данных для регулярной работы
- создаётся по умолчанию
- хорошая практика - также не использовать
- но и не удалять - иногда нужна для различных утилит

Соответственно для своих целей создаём свою БД используя DDL⁴⁷ create database⁴⁸:

CREATE DATABASE book;

Если теперь посмотреть **содержимое** каталога **base**, увидим ещё один новый каталог с нашей базой данных. Теперь поговорим о табличных пространствах.

Табличное пространство это:

- отдельный каталог с точки зрения файловой системы
- лучше делать отдельную файловую систему
- одно табличное пространство может использоваться несколькими базами данных
- каждой базе данных можно назначить как табличное пространство по умолчанию, так и в каждом конкретном случае указывать в каком табличном пространстве будет расположен конкретный объект внутри базы данных

По умолчанию есть два табличных пространства. Давайте на них посмотрим:

SELECT * FROM pg_tablespace;

```
postgres=# select * from pg_tablespace;
 oid | spcname | spcowner | spcacl | spcoptions
-----+-----+-----+-----+-----
 1663 | pg_default |      10 |      | 
 1664 | pg_global  |      10 |      | 
(2 rows)
```

Новые табличные пространства по умолчанию создаются в каталоге **\$PGDATA/pg_tblspc**. Также можем создать свой каталог и создать своё табличное пространство, указав путь к нашему каталогу. Попробуем на практике.

⁴⁷ DDL URL: https://en.wikipedia.org/wiki/Data_definition_language (дата обращения 10.03.2021) [5]

⁴⁸ Create database URL: <https://www.postgresql.org/docs/13/sql-createdatabase.html> (дата обращения 10.03.2021) [6]

Создадим каталог из-под суперпользователя Линукс **root**, для этого выйдем из-под пользователя postgres и зайдём под рутом. Поменяем его владельца на пользователя Линукс **postgres**:

```
exit  
sudo su  
sudo mkdir /home/postgres  
sudo chown postgres /home/postgres
```

Переключимся на пользователя Линукс postgres и в новом месте создадим каталог для нашего табличного пространства:

```
sudo su postgres  
cd /home/postgres  
mkdir tmptblspc
```

Теперь в утилите **psql** создадим само табличное пространство, где укажем путь ко вновь созданному каталогу:

```
psql  
CREATE TABLESPACE 49 ts location '/home/postgres/tmptblspc';
```

Посмотрим на список табличных пространств:

```
\db
```

```
postgres=# \db
```

List of tablespaces		
Name	Owner	Location
-----+-----+-----		
pg_default	postgres	
pg_global	postgres	
ts	postgres	/home/postgres/tmptblspc
(3 rows)		

Создадим базу данных с настройкой нового табличного пространства по умолчанию:

```
CREATE DATABASE app TABLESPACE ts;
```

Подключимся к созданной БД:

```
\c app
```

Чтобы посмотреть дефолтный tablespace, выполним:

```
V+
```

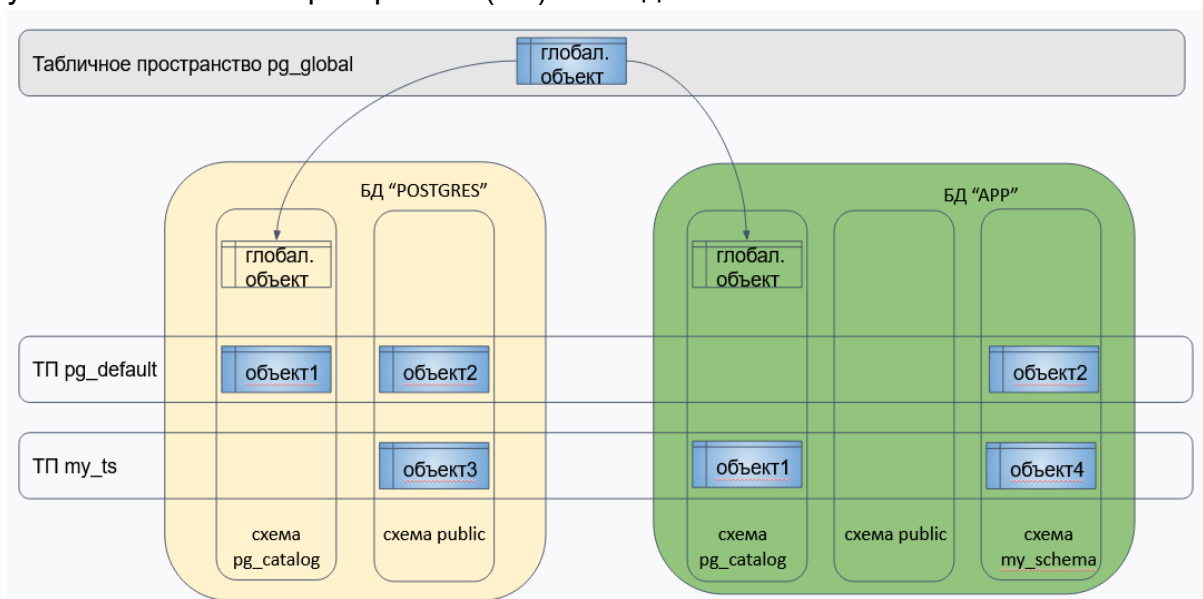
⁴⁹ Create tablespace URL: <https://www.postgresql.org/docs/current/sql-createtablespace.html> (дата обращения 10.03.2021) [7]

List of databases									
Name	Owner	Encoding	Collate	Ctype	Access privileges	Size	Tablespace	Description	
app	postgres	UTF8	C.UTF-8	C.UTF-8		7901 kB	ts		
book	postgres	UTF8	C.UTF-8	C.UTF-8		7901 kB	pg_default		
postgres	postgres	UTF8	C.UTF-8	C.UTF-8		7909 kB	pg_default		default administrative
template0	postgres	UTF8	C.UTF-8	C.UTF-8	=c/postgres	7753 kB	pg_default		unmodifiable empty
template1	postgres	UTF8	C.UTF-8	C.UTF-8	=c/postgres	7901 kB	pg_default		default template for new databases
(5 rows)									
(END)									

Чтобы выйти из полноэкранного режима, нужно нажать кнопку **q**.

Без конкретного указания табличного пространства при создании таблицы она автоматически будет расположена в tablespace **ts**. Конечно, можно точно указать, где конкретно хранить таблицу.

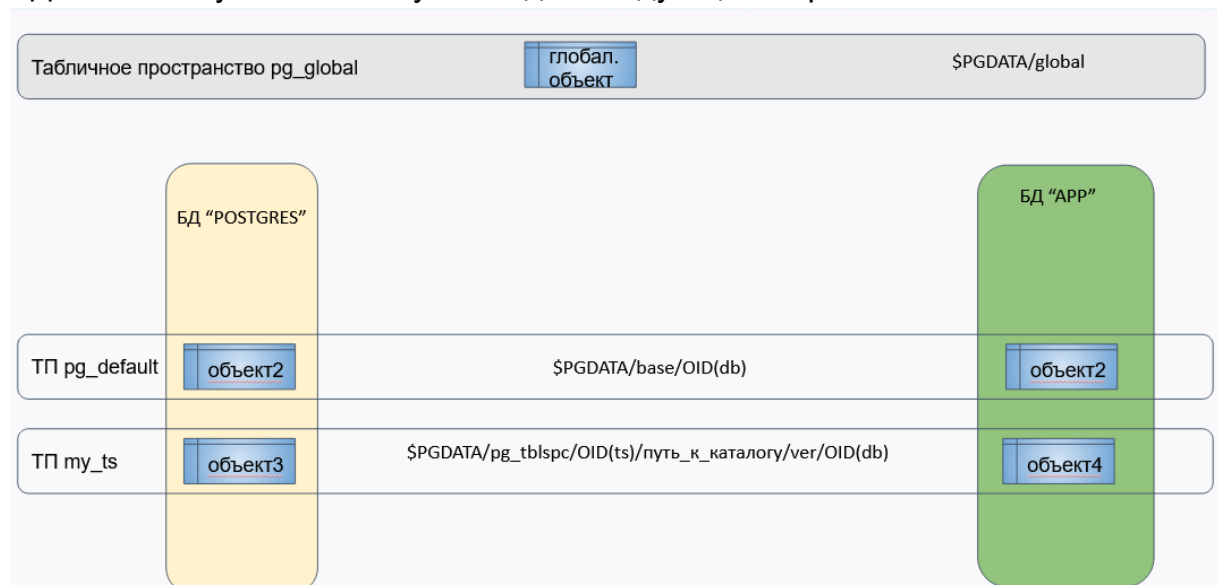
Вернёмся немного к теории, а именно как хранятся таблицы физически с учётом табличных пространств (ТП) и баз данных:



Видим, что объект может принадлежать только к одному табличному пространству (**pg_default** или другое), одной базе данных (**appdb** или **postgres**) и одной схеме (у каждой БД свой набор схем, про них будем говорить в теме про логическое устройство Постгреса). И только глобальные/системные объекты хранятся отдельно в табличном пространстве **pg_global** и присутствуют логически в каждой БД. При этом, одному объекту в базе данных может соответствовать от одного и более файлов в зависимости от типа объекта и его размера.

Конечно, можно перемещать объект между табличными пространствами, но нужно понимать, что это будет физическое перемещение файлов между каталогами, и на это время доступа к объекту не будет.

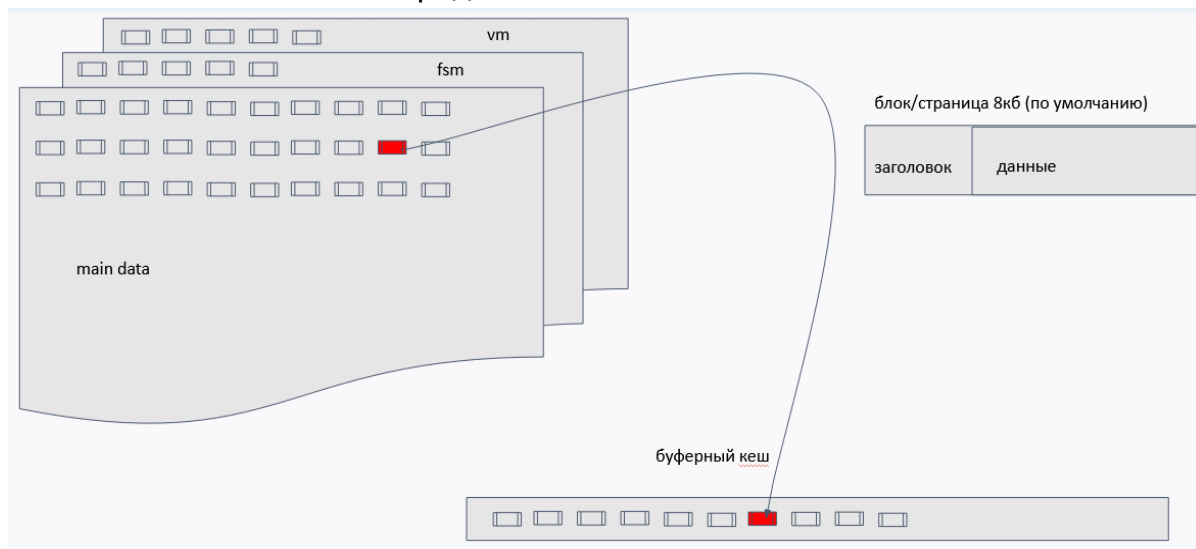
Также внутри каталога с табличным пространством расположены каталоги с базами данных. При этом используются также OID, только уже для БД. Полный путь к объекту выглядит следующим образом:



Внутри каталога с БД уже расположены непосредственно файлы объектов. Для каждой таблицы создаётся до трёх файлов (если размер сегмента больше 1 Гб, то будут созданы аналогичные файлы с добавлением номера сегмента .1 .2 и т.д.):

- файл с данными - OID таблицы
- файл со свободными блоками - OID_fsm (**free space map**)
 - отмечает свободное пространство в страницах после очистки
 - используется при вставке новых версий строк
 - существует для всех объектов
- файл с таблицей видимости - OID_vm (**visibility map**)
 - отмечает страницы, на которых все версии строк видны во всех снимках
 - используется для оптимизации работы процесса очистки и ускорения индексного доступа
 - существует только для таблиц
 - иными словами, это страницы, которые давно не изменялись и успели полностью очиститься от неактуальных версий

Схематично можно представить себе это так:



Обратим внимание, что работа с данными идёт не побайтно, а блоками/страницами по 8 Кб⁵⁰ (размер задаётся при инициализации кластера, обычно никто не меняет). Нужные данные извлекаются из файла и загружаются в буферный кэш постранично (более подробно, как в Постгресе организована работа со слоями и кэшем, будем разбирать в следующих темах).

Необходимо заметить, что вся строка нашей таблицы должна помещаться как раз в 8 Кб, и если она превышает этот размер, то используется специальная TOAST-таблица для хранения больших строк. Тут есть свои тонкости:

- используется схема **pg_toast**
- поддерживается собственным индексом
- читается только при обращении к «длинному» атрибуту
- **стоит задуматься, когда пишем `select * from ...`**
- собственная версионность (если при обновлении toast-часть не меняется, то и не будет создана новая версия toast-части)
- работает прозрачно для приложения

⁵⁰ Кб - килобайты

Собственно, в файловой системе файлы таблиц выглядят следующим образом:

```
postgres@postgres13:/home/postgres/tmp/tblspc/PG_13_202007201/16391$ ls -l
total 7912
-rw----- 1 postgres postgres 8192 Mar 10 14:18 112
-rw----- 1 postgres postgres 8192 Mar 10 14:18 113
-rw----- 1 postgres postgres 81920 Mar 10 14:18 1247
-rw----- 1 postgres postgres 24576 Mar 10 14:18 1247_fsm
-rw----- 1 postgres postgres 8192 Mar 10 14:18 1247_vm
-rw----- 1 postgres postgres 434176 Mar 10 14:18 1249
-rw----- 1 postgres postgres 24576 Mar 10 14:18 1249_fsm
-rw----- 1 postgres postgres 8192 Mar 10 14:18 1249_vm
-rw----- 1 postgres postgres 663552 Mar 10 14:18 1255
-rw----- 1 postgres postgres 24576 Mar 10 14:18 1255_fsm
-rw----- 1 postgres postgres 8192 Mar 10 14:18 1255_vm
-rw----- 1 postgres postgres 106496 Mar 10 14:18 1259
```

Видим, что размеры файлов кратны 8 Кб.

Давайте создадим свои таблицы в нашем новом табличном пространстве, попробуем потом на существующей таблице его изменить.

Создадим таблицу в дефолтном табличном пространстве ts:

CREATE TABLE test (i int);

Создадим таблицу в конкретном табличном пространстве:

CREATE TABLE test2 (i int) TABLESPACE pg_default;

Посмотрим, кто где создан:

SELECT tablename, tablespace FROM pg_tables WHERE schemaname = 'public';

```
app=# select tablename, tablespace from pg_tables where schemaname = 'public';
tablename | tablespace
-----+-----
test      |
test2     | pg_default
(2 rows)
```

Необходимо отметить, что для таблицы test в строке tablespace видим пустое значение - это означает, что таблица создана в дефолтном для этой базы данных табличном пространстве.

Изменим табличное пространство для таблицы test:

ALTER TABLE test SET tablespace pg_default;

```
app=# alter table test set tablespace pg_default;
ALTER TABLE
app=# select tablename, tablespace from pg_tables where schemaname = 'public';
tablename | tablespace
-----+-----
test2     | pg_default
test      | pg_default
(2 rows)
```

Также можем посмотреть, где лежит таблица:

SELECT pg_relation_filepath('test2');

Давайте теперь посмотрим на размеры таблиц, баз данных и табличных пространств изнутри Постгреса.

Узнать размер, занимаемый базой данных и объектами в ней, можно с помощью ряда функций:

SELECT pg_database_size('app');

```
app=# SELECT pg_database_size('app');
pg_database_size
-----
      8090159
(1 row)
```

Для упрощения восприятия возможно вывести число в отформатированном виде:

SELECT pg_size_pretty(pg_database_size('app'));

```
app=# SELECT pg_size_pretty(pg_database_size('app'));
pg_size_pretty
-----
      7901 kB
(1 row)
```

Полный размер таблицы (вместе со всеми индексами):

SELECT pg_size_pretty(pg_total_relation_size('test2'));

```
app=# SELECT pg_size_pretty(pg_total_relation_size('test2'));
pg_size_pretty
-----
      0 bytes
(1 row)
```

Остальные по аналогии без скриншотов - отдельно размер таблицы:

SELECT pg_size_pretty(pg_table_size('test2'));

И индексов:

SELECT pg_size_pretty(pg_indexes_size('test2'));

При желании можно узнать и размер отдельных слоёв таблицы, например:

SELECT pg_size_pretty(pg_relation_size('test2','vm'));

Размер табличного пространства показывает другая функция:

SELECT pg_size_pretty(pg_tablespace_size('ts'));

Общие рекомендации по использованию табличных пространств:

- не хранить данные в корневой файловой системе
- отдельная файловая система для каждого табличного пространства
- разделяя БД на табличные пространства - мы распараллеливаем файловую обработку и ускоряем БД в целом

- в случае внешнего файлового хранилища - отдельный каталог для каждого табличного пространства
- файловые системы EXT3/4 и XFS наиболее популярны

Мы рассмотрели, как физически расположены и хранятся конфиги и файлы БД.

Давайте теперь посмотрим, как Постгрес выглядит в процессах:

Первый процесс Постгреса - **postgres server process**:

- называется **postgres**
- запускается при старте сервиса
- порождает все остальные процессы используя клон процесса - **fork**
- создаёт shared memory
- слушает TCP и Unix socket

Остальные порождаемые им процессы это:

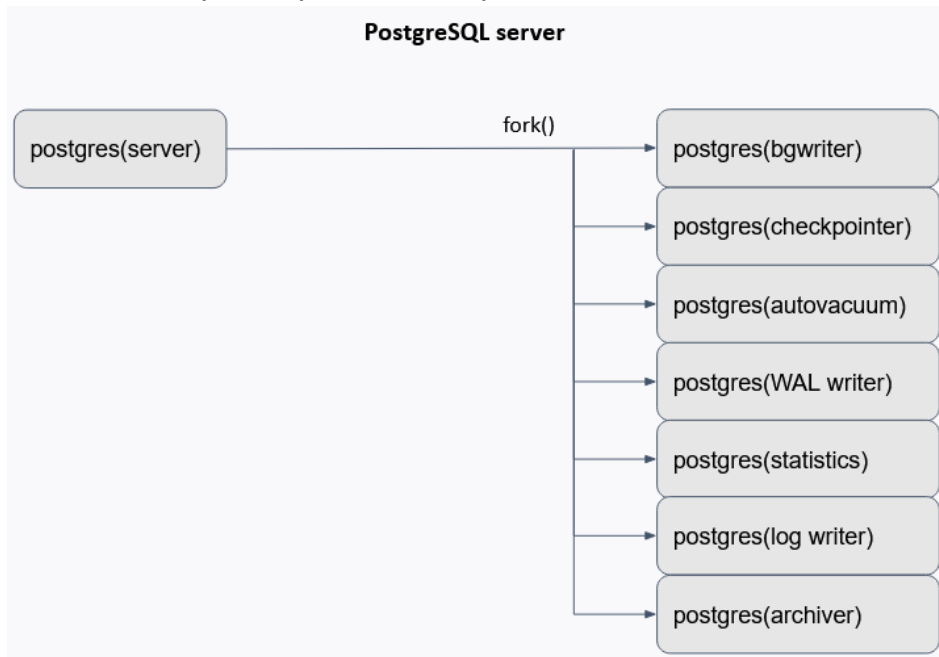
backend processes

- сейчас тоже называется **postgres**
- запускается основным процессом Постгреса
- обслуживает сессию
- работает, пока сессия активна
- максимальное количество определяется параметром **max_connections** (по умолчанию 100)

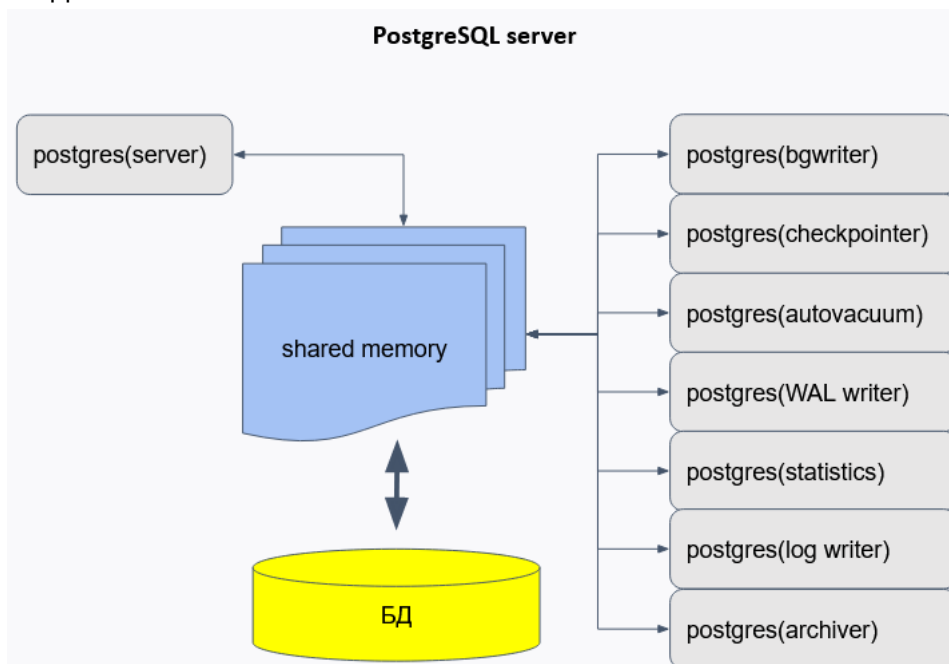
background processes:

- запускаются основным процессом Постгреса при старте сервиса
- выделенная роль у каждого процесса (будем рассматривать в дальнейших главах)
 - **logger** - запись сообщений в лог файл
 - **checkpointer** - запись грязных страниц из buffer cache на диск при наступлении checkpoint
 - **bgwriter** - проактивная запись грязных страниц из buffer cache на диск
 - **walwriter** - запись WAL buffer в WAL file
 - **autovacuum** - периодический запуск autovacuum
 - **archiver** - архивация и репликация WAL
 - **statscollector** - сбор статистики использования по сессиям и таблицам

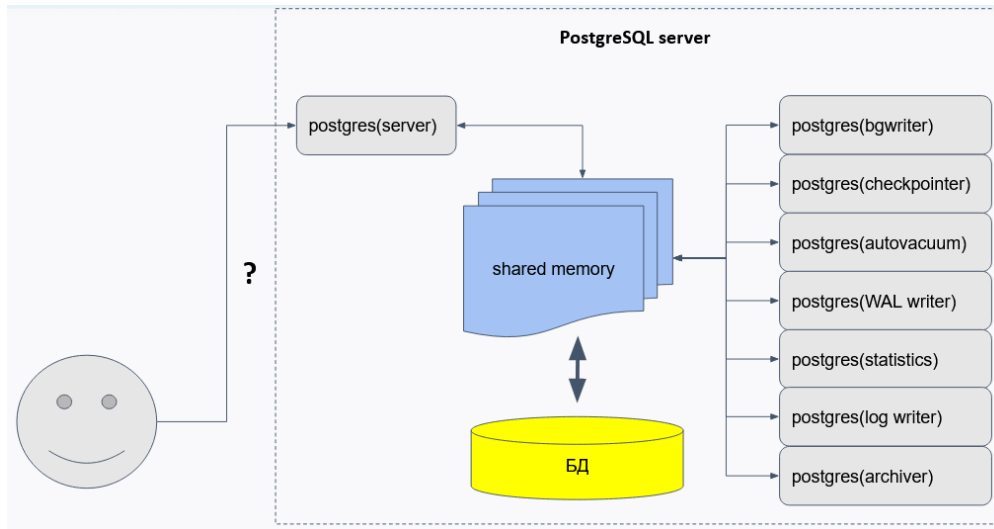
А теперь подробно, что происходит, схематично:



После того, как мы клонировали процессы, они начинают выполнять каждый свою задачу. Далее создаётся общая разделяемая память, которую все эти процессы совместно используют и через неё уже идёт общение с файлами на дисках:

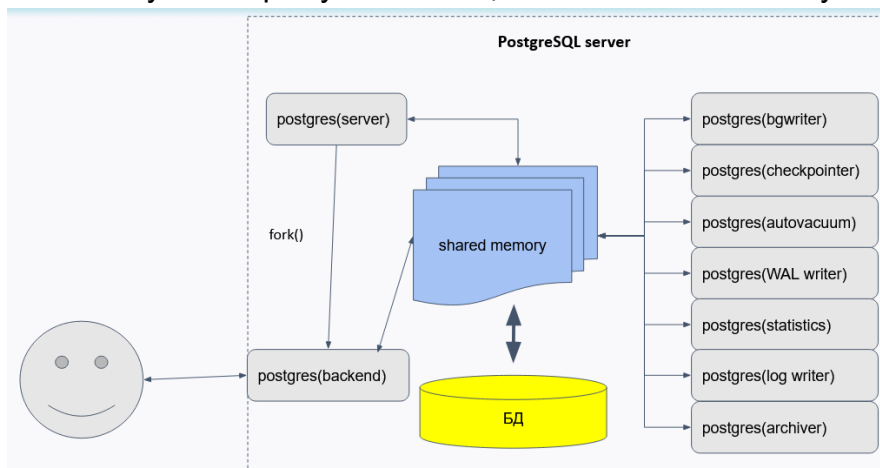


База данных готова к приёму подключений. Пользователь подключается к БД:



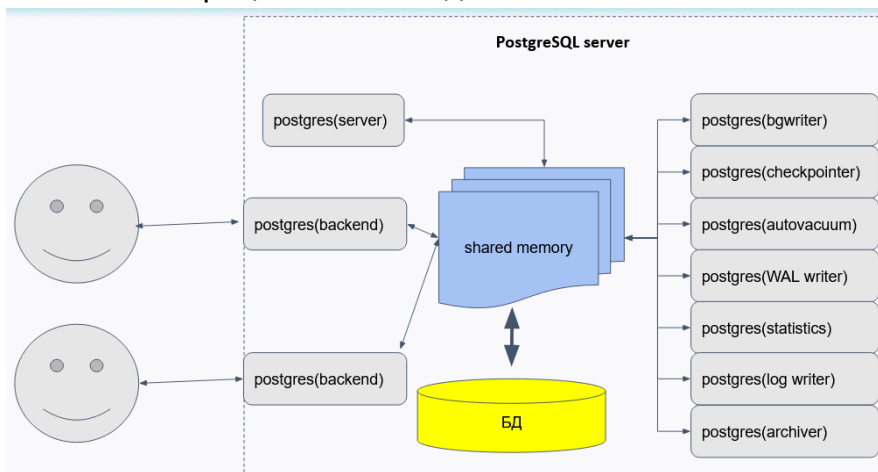
И что, как вы думаете, происходит?

Происходит **fork** процесса `postgres`, и дальше этот процесс **postgres backend** уже напрямую сам общается с `shared memory`:

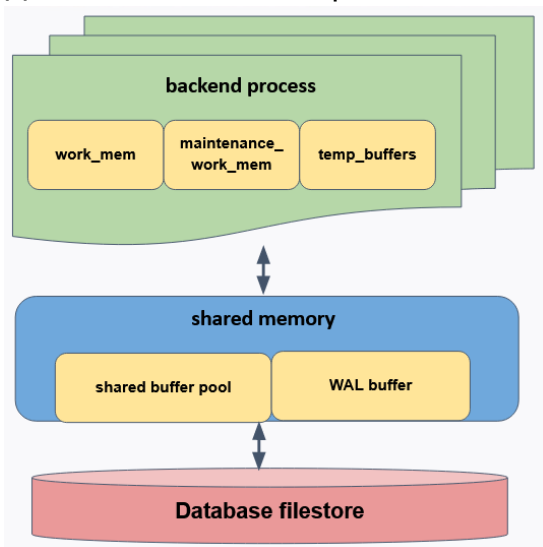


И вот приходит ещё один пользователь. И что теперь?

Снова клон процесса. На каждого пользователя свой бэкэнд-процесс:



Для каждого бэкэнд-процесса выделяется своя память:



work_mem (4MB)

- эта память используется на этапе выполнения запроса для сортировок строк, например ORDER BY и DISTINCT

maintenance_work_mem (64MB)

- используется служебными операциями типа VACUUM и REINDEX
- выделяется только при наличии таких команд в сессии

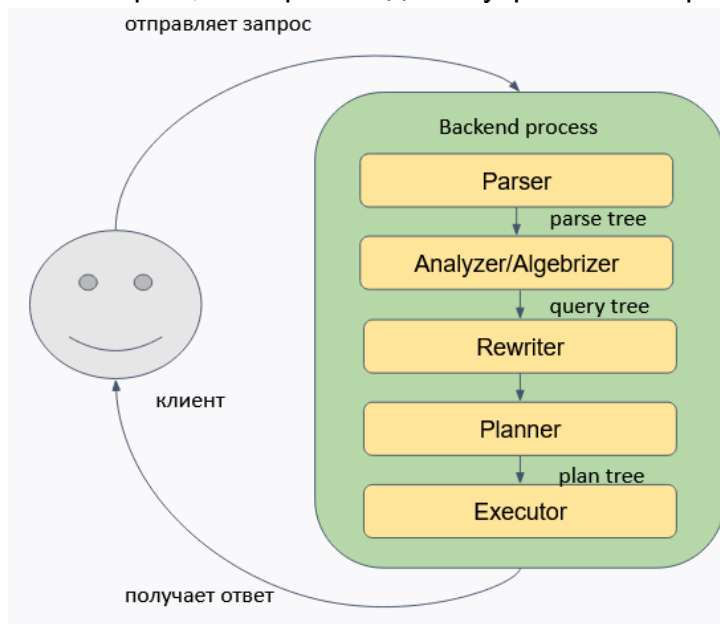
temp_buffers (8MB)

- используется на этапе выполнения для хранения временных таблиц

Все эти параметры настраиваются в зависимости от потребностей.

Shared buffer pool и **WAL buffer** будем рассматривать в дальнейших главах.

Рассмотрим, что происходит внутри сессии при запросе:



При выполнении запроса происходят следующие этапы:

- **Parser** - строится дерево парсинга
- **Analyser/Algebrizer** - строится дерево запросов
- **Rewriter** - переписываем исходный запрос
- **Planner** - строим план выполнения
- **Executor** - и только здесь выполняем запрос

Таким образом вы теперь понимаете, что процедура установки соединения в Постгресе очень дорогая. Именно поэтому рекомендуется использовать пулы подключений, например, **pgpool**⁵¹.

А теперь, раз заговорили про подключения, настроим доступ к нашему кластеру извне.

Так как по умолчанию доступ к Постгресу открыт только с локалхоста⁵², при попытке подключения к ВМ postgres13 по внешнему IP-адресу, например, с локальной машины, используя командную утилиту **psql** (её рассмотрим подробно в следующей главе), ожидаемо, результата не получим:

```
postgres13 us-central1-a e2-medium 10.128.0.13 34.66.131.159 RUNNING
aeugene@Aeuge:/mnt/c/Users/arist$ psql -h 34.66.131.159 -U postgres -W
Password:
psql: error: could not connect to server: could not connect to server: Connection refused
Is the server running on host "34.66.131.159" and accepting
TCP/IP connections on port 5432?
```

⁵¹ Pgpool URL: https://www.pgpool.net/mediawiki/index.php/Main_Page (дата обращения 10.03.2021) [8]

⁵² Localhost или 127.0.0.1 URL: <https://en.wikipedia.org/wiki/Localhost> (дата обращения 10.03.2021) [9]

Посмотрим на открытые соединения:

netstat -a|grep post

```
aeugene@postgres13:~$ netstat -a|grep post
tcp        0      0 localhost:postgresql 0.0.0.0:*          LISTEN
tcp        0      0 postgres13.us-cen:52676 142.250.128.95:https ESTABLISHED
tcp        0      0 postgres13.us-centr:ssh 109-252-76-78.nat:14346 ESTABLISHED
tcp        0      0 postgres13.us-cen:44860 metadata.google.in:http ESTABLISHED
udp        0      0 postgres13.us-ce:bootpc 0.0.0.0:*
unix 2      [ ACC ]     STREAM  LISTENING   21568      /var/run/postgresql/.s.PGSQL.5432
```

Видим, что слушаем (LISTENING) мы только localhost.

Для открытия доступа извне к кластеру нужно сделать ряд изменений в настройках Постгреса:

1. Включаем **listener** в **postgresql.conf**, раскомментируем соответствующую строчку **убрав решётку #**

sudo nano /etc/postgresql/13/main/postgresql.conf

`listen_addresses = '*'` *# IP адреса, на которых Постгрес принимает сетевые подключения, например, localhost, 10.*.*.*.*
*Установив значение * мы откроем доступ на всех сетевых адаптерах, но так делать на рабочих проектах не стоит.*

Второй вариант через утилиту **psql**, при этом изменение попадёт в файл **postgresql.auto.conf**:

SHOW listen_addresses;

ALTER SYSTEM SET listen_addresses = '*';

Третий вариант рассмотрим ближе к концу книги, когда уже будем профессионалами.

2. Включаем вход по паролю в **pg_hba.conf** и меняем маску подсети, откуда будет разрешён доступ к нашему кластеру:

sudo nano /etc/postgresql/13/main/pg_hba.conf:

```
host all all 0.0.0.0/0 md5
```

*Здесь мы открыли доступ везде, установив маску 0.0.0.0/0, так делать не стоит и желательно указывать маску максимально узко. Также мы разрешили доступ от имени всех пользователей **all** - желательно тоже подходить к этому вопросу более тщательно.*

*Обязательно указываем метод шифрования пароля **md5** - тогда он будет передан в зашифрованном виде. Также можно просто указать **password** - пароль будет передан в открытом виде и может быть перехвачен злоумышленником.*

3. Не забываем добавлять порт в **Google VPC** - рассматривали в прошлой главе.

4. Задаём пароль пользователю СУБД postgres через утилиту psql:
ALTER USER postgres PASSWORD 'Postgres123#';

Второй вариант используя встроенную команду psql:
\password

5. Перегружаем сервер через утилиту pg_ctlcluster:
sudo pg_ctlcluster 13 main restart

Протестируем доступ с ноутбука:

psql -h 34.66.131.159 -U postgres -W

```
aeugene@Aeuge:/mnt/c/Users/arist$ psql -h 34.66.131.159 -U postgres -W
Password:
psql (12.4 (Ubuntu 12.4-1.pgdg18.04+1), server 13.2 (Ubuntu 13.2-1.pgdg20.10+1))
WARNING: psql major version 12, server major version 13.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=# |
```

Вуаля, доступ получен.

В этой главе мы рассмотрели физический уровень - как и в каких файлах хранятся данные, как устроен Постгрес в физических процессах и как организовать физический доступ к кластеру Постгреса извне. В следующей главе рассмотрим интерактивную утилиту `psql`, с помощью которой можно полноценно работать с PostgreSQL.

** Задача на закрепление материала:*

Памяти у инстанса 4 Gb (периодически приходил OOM killer⁵³)

`max_connections = 1000` # (change requires restart)

`shared_buffers = 6GB` # min 128kB

`work_mem = 16MB` # min 64kB

`maintenance_work_mem = 256MB` # min 1MB

Что не так с параметрами?

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/02/physical_level.txt

⁵³ Out of Memory Killer URL: <https://habr.com/ru/company/southbridge/blog/464245/> (дата обращения 10.03.2021) [10]

3. Работа с консольной утилитой psql

psql⁵⁴ - интерактивный терминал Постгрес. Это так называемая **cli - command line interface**. Позволяет полноценно общаться с кластером Постгреса.

Полный синтаксис вызова psql смотреть не будем, так как если посмотреть помощь - **info psql**, то там больше 2000 строк.

Рассмотрим основные опции:

- h - хост, к которому подключаемся
- U - пользователь, под которым заходим
- W - интерактивный ввод пароля
- p - порт кластера Постгреса
- d - база данных для подключения

После подключения к БД видим справочную информацию:

```
aeugene@Aeuge:/mnt/c/Users/arist$ psql -h 34.66.131.159 -U postgres -W
Password:
psql (12.4 (Ubuntu 12.4-1.pgdg18.04+1), server 13.2 (Ubuntu 13.2-1.pgdg20.10+1))
WARNING: psql major version 12, server major version 13.
        Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.
```

- версию сервера 13.2
- ОС, которую использует сервер Ubuntu 20.10
- версию клиента для подключения 12.4
- версию ОС клиента - Ubuntu 18.04
- наше подключение использует шифрованное соединение **SSL**⁵⁵ по протоколу **TLS**⁵⁶

Далее видим приглашение для ввода **#**. Можно использовать различные варианты команд, как встроенных в psql, так и сами команды SQL:

- \? список команд psql
- \? **variables** переменные psql
- \h[elp] список команд SQL
- \h **команда** синтаксис команды SQL
- \q выход (до 11 версии), сейчас можно использовать команду **exit**

⁵⁴ psql URL: <https://www.postgresql.org/docs/13/app-psql.html> (дата обращения 10.03.2021) [1]

⁵⁵ SSL URL: https://en.wikipedia.org/wiki/Secure_Sockets_Layer (дата обращения 10.03.2021) [2]

⁵⁶ TLS URL: https://en.wikipedia.org/wiki/Transport_Layer_Security (дата обращения 10.03.2021) [3]

Приведу здесь небольшой список наиболее полезных и часто встречающихся команд:

- **\l** - список баз данных
- **\dp** (или **\z**) - список таблиц, представлений, последовательностей, прав доступа к ним
- **\di** - список индексов
- **\dt** - список таблиц
- **\dt+** - список всех таблиц с описанием
- **\dt *s*** - список всех таблиц, содержащих s в имени
- **\d+** - описание таблицы
- **\d "table_name"** - описание таблицы
- **\du** - список пользователей
- **\pset** - команда настройки параметров форматирования
- **\echo** - выводит сообщение
- **\set** - устанавливает значение переменной среды. Без параметров выводит список текущих переменных (**\unset** - удаляет)
- **\i** - запуск команды из внешнего файла, например **\i /home/test/my.sql**

Давайте более подробно остановимся на запуске команд из внешнего файла. Для этого сначала зайдём в Линукс под пользователем postgres:

sudo su postgres

Подключимся к созданной нами базе данных **app**:

\c app

Посмотрим, какие таблицы в ней существуют:

\dt

Выполним простейшую SQL-команду:

SELECT * FROM test;

```
book=# \c app
Password:
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
You are now connected to database "app" as user "postgres".
app=# \dt
      List of relations
 Schema | Name  | Type  | Owner
-----+-----+-----+-----
 public | test  | table | postgres
 public | test2 | table | postgres
(2 rows)

app=# select * from test;
 i
---
(0 rows)

app=#
```

Затем выйдем из psql, запишем эти команды в файл через редактор **nano**. При этом **файлы будем создавать в каталоге, куда есть доступ к линукс-пользователя postgres**, так как **psql запускаем** под этим пользователем и, соответственно, нам нужен доступ к файловой системе именно под линукс-пользователем postgres.

Выходим из psql:

exit

После этого переходим в домашний каталог линукс-пользователя postgres:

cd \$HOME

Посмотрим, где он расположен:

pwd

Видим, что он как раз расположен по адресу **/var/lib/postgresql**, где и находятся файлы нашего кластера по умолчанию. Посмотрим содержимое каталога:

ls -l

Видим каталог **13**, в котором, как помним, есть подкаталоги **main** и **main2** с данными кластеров.

```
postgres=# exit
postgres@postgres13:/home/aeugene$ cd $HOME
postgres@postgres13:~$ pwd
/var/lib/postgresql
postgres@postgres13:~$ ls -l
total 4
drwxr-xr-x 4 postgres postgres 4096 Mar 10 12:18 13
postgres@postgres13:~$ mkdir scripts
postgres@postgres13:~$ cd scripts/
postgres@postgres13:~/scripts$ nano select.sql
postgres@postgres13:~/scripts$
```

Создадим каталог со скриптами:

mkdir scripts

Перейдём в него:

cd scripts

Создадим файл psql.txt:

nano select.sql

Запишем туда наши три команды.

Зайдём обратно в psql и выполним скрипт, используя команду **psql \i**:

\i /var/lib/postgresql/scripts/select.sql

```
postgres=# \i /var/lib/postgresql/scripts/select.sql
You are now connected to database "app" as user "postgres".
      List of relations
Schema | Name  | Type  | Owner
-----+-----+-----+-----
public | test  | table | postgres
public | test2 | table | postgres
(2 rows)

i
---
(0 rows)

app=#
```

Видим, что наш скрипт прекрасно отработал. Таким образом можем заранее заготавливать нужные нам скрипты или подготавливать миграции данных.

При **наборе команд в psql** нужно обращать внимание на правильность ввода команд. Об этом свидетельствует знак "=" после имени БД, к которой мы подключены, и перед #.

Например, если забыли добавить точку с запятой в конец SQL-команды, то оболочка psql будет ждать окончания ввода команды для дальнейшего запуска парсера и т.д. Увидеть это можно, обратив внимание на **изменение знака равно на минус**:

```
app=# select * from test
app=#
app=#
app=# ;
i
---
(0 rows)

app=#
```

В данном случае psql ждал окончания ввода команды - нужно было или добавить точку с запятой или **прервать ввод команды нажав Control + C**.

Важный момент, что при запуске psql выполняются команды, записанные в двух файлах - общесистемном и пользовательском. Общий системный файл называется **psqlrc** и располагается в каталоге по умолчанию: **/etc/postgresql-common** и **/usr/local/pgsql/etc** при обычной сборке из исходных кодов.

Расположение этого каталога можно узнать командой:

pg_config --sysconfdir

```
postgres@postgres13:~$ pg_config --sysconfdir
/etc/postgresql-common
postgres@postgres13:~$
```

Обратите внимание на \$ в приглашении командной строки - это командная строка Линукс. Внутри psql приглашение в виде # для рута и > для обычного пользователя.

Пользовательский файл находится в домашнем каталоге пользователя ОС и называется **.psqlrc**. Его расположение можно изменить, задав переменную окружения PSQLRC.

В эти файлы можно записать команды, настраивающие **psql**. Например, изменить приглашение, включить вывод времени выполнения команд и т.п.

История вводимых команд сохраняется в файле **.psql_history** в домашнем каталоге пользователя. По умолчанию хранится 500 последних команд, это число можно изменить переменной psql HISTSIZE.

Также зачастую в ситуации, когда много колонок в нашей таблице, вывод на экран очень неинформативен:

SELECT * FROM pg_stat_activity;

datid	datname	pid	leader_pid	usesysid	username	application_name	client_addr	client_hostname	client_port	backend_start	xact_start	query_start	state_change		
										wait_event_type	wait_event	state	backend_xid	backend_xmin	query
		684													
		2021-03-12 12:12:45.457754+00								Activity	AutoVacuumMain				
		autovacuum launcher		10	postgres										
		686													
		2021-03-12 12:12:45.458728+00								Activity	LogicalLauncherMain				
		logical replication launcher		10	postgres	psql									
13445	postgres	2939		10	postgres	psql									
-1		2021-03-12 13:16:54.634896+00		2021-03-12 13:16:56.471017+00		2021-03-12 13:16:56.471017+00		2021-03-12 13:16:56.471021+00			active		498	select * from pg_stat_activ	ity;
		client backend													

Можем включить расширенный вывод информации - вертикальный вывод колонок:

\x

SELECT * FROM pg_stat_activity;

-[RECORD 1]-----	
datid	
datname	
pid	684
leader_pid	
usesysid	
username	
application_name	
client_addr	
client_hostname	
client_port	
backend_start	2021-03-12 12:12:45.457754+00
xact_start	
query_start	
state_change	
wait_event_type	Activity
wait_event	AutoVacuumMain
state	
backend_xid	
backend_xmin	
query	
backend_type	autovacuum launcher
-[RECORD 2]-----	
datid	

Согласитесь, намного более наглядный вывод.

Повторный ввод команды \x отменит расширенный вывод:

\x

Также есть вариант указания **\gx в конце строки** вместо точки с запятой - получим расширенный вывод для этой конкретной команды.

В этой главе мы рассмотрели основы использования консольной утилиты psql. Переходим к изучению ACID, MVCC и всё, что с этим связано.

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/03/psql.txt

4. ACID & MVCC. Vacuum и autovacuum

Реляционная теория и SQL позволяет абстрагироваться от конкретной реализации СУБД, но есть одна непростая проблема: ***как обеспечить параллельную работу множества сессий (concurrency), которые модифицируют данные, так, чтобы они не мешали друг другу ни с точки зрения чтения, ни с точки зрения записи и обеспечивали целостность данных (consistency) и их надежность (durability)?***

Ответ - транзакционные системы OLTP⁵⁷ - Online Transaction Processing. Они отвечают следующему принципу **ACID**⁵⁸:

- **Atomicity** - атомарность
- **Consistency** - согласованность
- **Isolation** - изолированность
- **Durability** - долговечность

Соответственно, транзакция (**transaction**) это:

- множество операций, выполняемое приложением
- которое переводит базу данных из одного корректного состояния в другое корректное состояние (согласованность)
- при условии, что транзакция выполнена полностью (атомарность)
- и без помех со стороны других транзакций (изолированность)

Всё было бы хорошо, но что делать с блокировками и сбоями?

Для этого придуманы следующие принципы:

ARIES⁵⁹ - Algorithms for Recovery and Isolation Exploiting Semantics. Алгоритмы эффективного восстановления после сбоев.

Использует следующие механизмы:

- logging - логирование
- undo - сегменты отката транзакций
- redo - сегменты проигрывания транзакций после сбоя
- checkpoints - система контрольных точек

⁵⁷ OLTP URL: https://en.wikipedia.org/wiki/Online_transaction_processing (дата обращения 13.03.2021) [1]

⁵⁸ ACID URL: <https://en.wikipedia.org/wiki/ACID> (дата обращения 13.03.2021) [2]

⁵⁹ ARIES URL: https://en.wikipedia.org/wiki/Algorithms_for_Recovery_and_Isolation_Exploiting_Semantics (дата обращения 13.03.2021) [3]

MVCC⁶⁰ - Multiversion Concurrency Control. Конкурентный контроль мультиверсионности. Использует следующие механизмы:

- сору-on-write - создаётся копия старых данных при записи или модификации
- каждый пользователь работает со снимком БД
- вносимые пользователем изменения не видны другим до фиксации транзакции
- практически не использует блокировок (только одна - писатель блокирует писателя, если они пытаются работать с одной записью)
- или ручная блокировка при вызове команды select for update

Эти все механизмы рассмотрим в этой и последующих главах.

Механизм реализации MVCC в PostgreSQL.

Данные хранятся поблочно и посмотрим, из чего состоит один блок:

HeapTupleHeader data									
xmin	xmax	cmin	cmax	t_cid	t_ctid	t_infomask	t_infomask2	Null bitmap	Data

В самом начале идёт служебный блок, состоящий из пары идентификаторов транзакций по 4 байта, ряда информационных байтов, пары информационных масок и пустой битовой карты для будущего использования в следующих версиях Постгреса и, собственно, данных (более подробно будем рассматривать в следующих темах).

Если рассматривать подробно, то заголовок состоит из:

- **xmin** - идентификатор транзакции, которая создала данную версию записи
- **xmax** - идентификатор транзакции, которая удалила данную версию записи
- **cmin** - порядковый номер команды в транзакции, добавившей запись
- **cmax** - номер команды в транзакции, удалившей запись

⁶⁰ MVCC URL: https://en.wikipedia.org/wiki/Multiversion_concurrency_control (дата обращения 13.03.2021) [4]

Соответственно, когда мы в Постгрес выполняем операции вставки, обновления или удаления записи, происходят следующие изменения этих значений:

- **Insert** - добавляется новая запись с **xmin = txid_current()** и **xmax = 0**
- **Update** - в старой версии записи **xmax = txid_current()**, то есть делается **delete**, добавляется новая запись с **xmin = txid_current()** и **xmax = 0**, то есть делается **insert**
- **Delete** - в старой версии записи **xmax = txid_current()**

Получается, при добавлении записи у нас происходит две операции: старая запись помечается как удалённая, проставляется **xmax**, и происходит добавление новой записи. При удалении проставляется значение **xmax**, при этом физическое удаление данных не производится.

Для фиксации или отмены транзакции нам нужно рассмотреть ещё дополнительные атрибуты строки - **infomask** содержит ряд битов, определяющих свойства данной версии:

- **xmin_committed, xmin_aborted** - xmin подтверждён или отменён
- **xmax_committed, xmax_aborted** - xmax подтверждён или отменён
- **ctid** является ссылкой на следующую, более новую, версию той же строки. У самой новой, актуальной, версии строки **ctid** ссылается на саму эту версию. Номера **ctid** имеют вид (x,y): здесь x - номер страницы, y - порядковый номер указателя в массиве

Каждая транзакция может завершиться как **успешно** - после команды **commit**, для этого, в зависимости от типа операции, проставляются биты **xmin_committed, xmax_committed** - так и **неуспешно** из-за, например, ошибки доступа к данным или вызове команды **rollback**. При неуспешном завершении нам нужно откатить все изменения, выполненные этой транзакцией. При этом в случае с Постгресом нам практически не нужно ничего делать - будут установлены биты подсказки **xmin_aborted, xmax_aborted**. Сам номер xmax при этом остаётся в странице, но смотреть на него уже никто не будет. Так что операции как коммита, так и роллбэка транзакции в Постгресе одинаково быстры.

Посмотрим на практике, как выглядит изнутри служебная информация.

Номер текущей транзакции можно узнать командой:

SELECT txid_current();

```
postgres=# select txid_current();
 txid_current
-----
          499
(1 row)

postgres=# \c app
You are now connected to database "app" as user "postgres".
app=# select txid_current();
 txid_current
-----
          500
(1 row)

app=#
```

При этом обратите внимание, что при смене БД у нас старая транзакция завершается отменой (**rollback**) и начинается новая транзакция.

Добавим в нашу ранее созданную в предыдущей главе таблицу **test** три новых значения:

INSERT INTO test VALUES (10),(20),(30);

Посмотрим статистику по “живым” (актуальным) и “мёртвым” (старым) версиям в результате **update** или **delete** туплам (записям):

SELECT relname, n_live_tup, n_dead_tup, trunc(100*n_dead_tup/(n_live_tup+1))::float "ratio%", FROM pg_stat_user_tables WHERE relname = 'test';

```
app=# SELECT relname, n_live_tup, n_dead_tup, trunc(100*n_dead_tup/(n_live_tup+1))::float "ratio%" FROM pg_stat_user_tables WHERE relname = 'test';
 relname | n_live_tup | n_dead_tup | ratio%
-----+-----+-----+-----
 test   |          3 |           0 |      0
(1 row)
```

Теперь, если обновить одну строчку из нашего набора и посмотрим предыдущий запрос:

UPDATE test SET i = 40 WHERE i = 30;

```
app=# update test set i = 40 where i = 30;
UPDATE 1
app=# SELECT relname, n_live_tup, n_dead_tup, trunc(100*n_dead_tup/(n_live_tup+1))::float "ratio%" FROM pg_stat_user_tables WHERE relname = 'test';
 relname | n_live_tup | n_dead_tup | ratio%
-----+-----+-----+-----
 test   |          3 |           1 |     25
(1 row)
```

Видим, что теперь запрос выдаст другую информацию.

Итого, имеем три актуальных записи и одну мёртвую, содержащую старую информацию в строке со значением 30.

Напрямую можно посмотреть служебную информацию только по пяти полям и только у актуальных записей:

SELECT xmin,xmax,cmin,cmax,ctid FROM test;

```
app=# select xmin,xmax,cmin,cmax,ctid from test;
 xmin | xmax | cmin | cmax | ctid
-----+-----+-----+-----+-----
  501  |    0 |    0  |    0  | (0,1)
  501  |    0 |    0  |    0  | (0,2)
  502  |    0 |    0  |    0  | (0,4)
(3 rows)
```

Здесь видим из колонки ctid, что у нас не видно запись со страницы под номером 3.

Для более подробного изучения нужно установить расширение Постгреса **pageinspect**:

CREATE EXTENSION pageinspect;

Посмотрим, какие функции оно содержит (ограничимся первыми на скриншоте):

\dx+

```
app=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
app=# \dx+
               Objects in extension "pageinspect"
               Object description
-----
function brin_metapage_info(bytea)
function brin_page_items(bytea,regclass)
function brin_page_type(bytea)
function brin_revmap_data(bytea)
function bt_metap(text)
function bt_page_items(bytea)
function bt_page_items(text,integer)
function bt_page_stats(text,integer)
function fsm_page_contents(bytea)
function get_raw_page(text,integer)
function get_raw_page(text,text,integer)
function gin_leafpage_items(bytea)
function gin_metapage_info(bytea)
function gin_page_opaque_info(bytea)
function hash_bitmap_info(regclass,bigint)
function hash_metapage_info(bytea)
function hash_page_items(bytea)
function hash_page_stats(bytea)
function hash_page_type(bytea)
function heap_page_item_attrs(bytea,regclass)
function heap_page_item_attrs(bytea,regclass,boolean)
function heap_page_items(bytea)
function heap_tuple_infomask_flags(integer,integer)
function page_checksum(bytea,integer)
```

Давайте для примера воспользуемся **heap_page_items** и **get_raw_page**:

SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid FROM heap_page_items(get_raw_page('test',0));

```
app=# select lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid from heap_page_items(get_raw_page('test',0));
 tuple | t_xmin | t_xmax | t_cid | t_ctid
-----+-----+-----+-----+-----
      1 |    501 |      0 |      0 | (0,1)
      2 |    501 |      0 |      0 | (0,2)
      3 |    501 |    502 |      0 | (0,4)
      4 |    502 |      0 |      0 | (0,4)
(4 rows)
```

Здесь получили доступ к сырой версии таблицы и видим третью строчку. А в ней видим проставленный **t_xmax** и ссылку на новую версию строки **t_ctid**.

Также можно посмотреть всё содержимое служебных полей командой:

SELECT * FROM heap_page_items(get_raw_page('test',0)) \gx

```
-[ RECORD 1 ]-----
lp           | 1
lp_off       | 8160
lp_flags     | 1
lp_len       | 28
t_xmin       | 501
t_xmax       | 0
t_field3     | 0
t_ctid       | (0,1)
t_infomask2  | 1
t_infomask   | 2304
t_hoff       | 24
t_bits       |
t_oid        |
t_data       | \x0a000000
-[ RECORD 2 ]-----
lp           | 2
lp_off       | 8128
lp_flags     | 1
lp_len       | 28
t_xmin       | 501
t_xmax       | 0
t_field3     | 0
t_ctid       | (0,2)
t_infomask2  | 1
t_infomask   | 2304
t_hoff       | 24
t_bits       |
t_oid        |
t_data       | \x14000000
```

Более подробную информацию смотреть уже труднее. Воспользуемся запросом:

SELECT '(0, '||lp||')' AS ctid,
CASE lp_flags
WHEN 0 THEN 'unused'
WHEN 1 THEN 'normal'
WHEN 2 THEN 'redirect to '||lp_off
WHEN 3 THEN 'dead'
END AS state,

```

t_xmin as xmin,
t_xmax as xmax,
(t_infomask & 256) > 0 AS xmin_committed,
(t_infomask & 512) > 0 AS xmin_aborted,
(t_infomask & 1024) > 0 AS xmax_committed,
(t_infomask & 2048) > 0 AS xmax_aborted,
t_ctid
FROM heap_page_items(get_raw_page('test',0)) lgx

```

-[RECORD 2]-+-----	
ctid	(0,2)
state	normal
xmin	501
xmax	0
xmin_committed	t
xmin_aborted	f
xmax_committed	f
xmax_aborted	t
t_ctid	(0,2)
-[RECORD 3]-+-----	
ctid	(0,3)
state	normal
xmin	501
xmax	502
xmin_committed	t
xmin_aborted	f
xmax_committed	t
xmax_aborted	f
t_ctid	(0,4)

Видим, что возле удалённой третьей записи стоит флаг **xmax_committed**, а у второй актуальной записи наоборот стоит флаг **xmax_aborted**. Соответственно, чтобы удалять мёртвые записи нужен какой-то механизм. И он называется вакуум. О нём сейчас и поговорим.

Vacuum⁶¹.

VACUUM высвобождает пространство, занимаемое мёртвыми строками. Соответственно чем больше таких операций, тем больше места занимают таблицы и индексы. Таким образом, периодически необходимо выполнять VACUUM, особенно для часто изменяемых таблиц.

Оператор VACUUM специфичен именно для MVCC PostgreSQL.

Без списка *таблиц и столбцов* команда VACUUM обрабатывает все таблицы и материализованные представления в текущей базе данных, на очистку которых текущий пользователь имеет право.

Команда VACUUM только делает пространство доступным для повторного использования. Эта форма команды может работать параллельно с

⁶¹ Vacuum URL: <https://www.postgresql.org/docs/13/sql-vacuum.html> (дата обращения 13.03.2021)
[5]

обычными операциями чтения и записи строк таблицы, так она **не требует исключительной блокировки**.

Посмотрим на самые важные параметры:

VERBOSE - выводит на экран отчёт об очистке для каждой таблицы.

ANALYZE - выполняет очистку (VACUUM), а затем анализ (ANALYZE) всех указанных таблиц. Это удобная комбинация для регулярного обслуживания БД. Команда `analyze` и принцип её работы у нас будут рассмотрены в дальнейших главах.

FULL - выбирает режим полной очистки, который еще и дефрагментирует пространство - соответственно выполняется гораздо дольше и **запрашивает исключительную блокировку таблицы**. Так как он записывает новую копию таблицы и не освобождает старую до завершения операции, то требует дополнительное место на диске.

SKIP_LOCKED - указывает, что команда VACUUM пропускает все строки с конфликтующими блокировками.

Соответственно, наблюдать за процессом очистки можно или в командной строке при выполнении **VACUUM VERBOSE**, или через системное представление **SELECT * FROM pg_stat_progress_vacuum;**

Пример использования:

Очистка одной таблицы `test`, проведение её анализа для оптимизатора и печать подробного отчёта о действиях операции очистки:

VACUUM (VERBOSE, ANALYZE) test;

Посмотрим, что теперь осталось в таблице:

SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid FROM heap_page_items(get_raw_page('test',0));

```
app=# VACUUM (VERBOSE, ANALYZE) test;
INFO: vacuuming "public.test"
INFO: "test": found 0 removable, 3 nonremovable row versions in 1 out of 1 pages
DETAIL: 0 dead row versions cannot be removed yet, oldest xmin: 504
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO: analyzing "public.test"
INFO: "test": scanned 1 of 1 pages, containing 3 live rows and 0 dead rows; 3 rows in sample, 3 estimated total rows
VACUUM
app=# select lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid from heap_page_items(get_raw_page('test',0));
 tuple | t_xmin | t_xmax | t_cid | t_ctid 
-----+-----+-----+-----+-----
      1 |      0 |      0 |      0 | (0,1)
      2 |      0 |      0 |      0 | (0,2)
      3 |      0 |      0 |      0 | 
      4 |     502 |      0 |      0 | (0,4)
(4 rows)
```

Очистка - отличный механизм, вопрос в том, как часто её вызывать.

Если очищать изменяющуюся таблицу слишком редко, она вырастет в размерах больше, чем хотелось бы. Кроме того, для очередной очистки может потребоваться несколько проходов по индексам, если изменений накопилось слишком много.

Если очищать таблицу слишком часто, то вместо полезной работы сервер будет постоянно заниматься обслуживанием - тоже нехорошо.

Также важно понимать, что запуск обычной очистки по расписанию никак не решает проблему, потому что нагрузка может изменяться со временем. Если таблица стала обновляться активней, то и очищать её надо чаще.

Более подробно про процесс Vacuum можно почитать в статье⁶² на Хабр⁶³.

Автоматическая очистка - как раз тот самый механизм, который позволяет запускать очистку в зависимости от активности изменений в таблицах. Итак:

Autovacuum⁶⁴.

Автовакуум управляется соответствующим **фоновым процессом Autovacuum launcher**. Т.е. он не работает всё время, а инициирует вызов вакуума в зависимости от настроек различных порогов срабатывания. Желательно настаивать максимально агрессивно.

Посмотрим на важные настройки⁶⁵:

log_autovacuum_min_duration - время выполнения автоочистки в мс, при превышении которого информация об этом действии записывается в жу По умолчанию стоит -1, что отключает журналирование действий автоочистки. При нулевом значении в журнале фиксируются все действия автоочистки.

autovacuum_max_workers - максимальное число процессов автоочистки (не считая процесса autovacuum_launcher), которые могут выполняться одновременно. По умолчанию 3.

autovacuum_naptime - минимальная задержка в секундах между двумя запусками автоочистки. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. По умолчанию задержка равна 1min.

Другие параметры влияют на то, как часто будет автовакуум заходить в таблицы для очистки. Про них можно почитать по ссылке выше.

⁶² MVCC-6. Очистка URL: <https://habr.com/ru/company/postgrespro/blog/452320/> (дата обращения 23.09.2021) [6]

⁶³ Хабр URL: <https://habr.com> (дата обращения 23.09.2021) [7]

⁶⁴ Autovacuum URL: <https://www.postgresql.org/docs/13/routine-vacuuming.html#AUTOVACUUM> (дата обращения 13.03.2021) [8]

⁶⁵ Параметры автовакуума URL: <https://www.postgresql.org/docs/13/runtime-config-autovacuum.html> (дата обращения 23.09.2021) [9]

Практически все эти параметры задаются только в postgresql.conf или в командной строке при запуске сервера. По разнице в настройке отдельных параметров будет целая тема чуть позже. Например, часть параметров можно мягко перезагрузить:

```
sudo pg_ctlcluster 13 main reload
```

Посмотрим на текущие настройки кластера:

```
SELECT name, setting, context, short_desc FROM pg_settings WHERE  
category LIKE '%Autovacuum%';
```

name	setting	context	short_desc
autovacuum	on	sighup	Starts the autovacuum subprocess.
autovacuum_analyze_scale_factor	0.1	sighup	Number of tuple inserts, updates, or deletes prior to analyze as a fraction of reltuples.
autovacuum_analyze_threshold	50	sighup	Minimum number of tuple inserts, updates, or deletes prior to analyze.
autovacuum_freeze_max_age	200000000	postmaster	Age at which to autovacuum a table to prevent transaction ID wraparound.
autovacuum_max_workers	3	postmaster	Sets the maximum number of simultaneously running autovacuum worker processes.
autovacuum_multixact_freeze_max_age	400000000	postmaster	Multixact age at which to autovacuum a table to prevent multixact wraparound.
autovacuum_naptime	60	sighup	Time to sleep between autovacuum runs.
autovacuum_vacuum_cost_delay	2	sighup	Vacuum cost delay in milliseconds, for autovacuum.
autovacuum_vacuum_cost_limit	-1	sighup	Vacuum cost amount available before napping, for autovacuum.
autovacuum_vacuum_insert_scale_factor	0.2	sighup	Number of tuple inserts prior to vacuum as a fraction of reltuples.
autovacuum_vacuum_insert_threshold	1000	sighup	Minimum number of tuple inserts prior to vacuum, or -1 to disable insert vacuums.
autovacuum_vacuum_scale_factor	0.2	sighup	Number of tuple updates or deletes prior to vacuum as a fraction of reltuples.
autovacuum_vacuum_threshold	50	sighup	Minimum number of tuple updates or deletes prior to vacuum.
(13 rows)			

Здесь видим такие настройки, как **autovacuum_freeze_max_age** и **autovacuum_multixact_freeze_max_age**.

Это ещё один аспект работы вакуума и автовакуума - они “замораживают” старые записи. Для чего это нужно сейчас и посмотрим.

Более подробно как работает Autovacuum можно посмотреть в статье⁶⁶ на habr.com.

⁶⁶ MVCC в PostgreSQL-8. Заморозка URL: <https://habr.com/ru/company/postgrespro/blog/452762/> (дата обращения 23.09.2021) [10]

Заморозка⁶⁷.

В Постгресе семантика транзакций зависит от возможности сравнения номеров идентификаторов транзакций (XID).

Версия строки, у которой XID добавившей её транзакции больше, чем XID текущей транзакции, относится «к будущему» и не должна быть видна в текущей транзакции.

Однако поскольку идентификаторы транзакций имеют ограниченный размер (32 бита), кластер, работающий долгое время (более 4 миллиардов транзакций) столкнётся с *зацикливанием идентификаторов транзакций*: счётчик XID дойдёт до максимального значения и прокрутится до нуля - внезапно транзакции, которые относились к прошлому, окажутся в будущем - это означает, что их результаты станут невидимыми. Для того, чтобы этого избежать, необходимо выполнять очистку для каждой таблицы в каждой базе данных как минимум единожды на два миллиарда транзакций.

Периодическое выполнение очистки решает эту проблему, потому что процедура **VACUUM** помечает строки как **замороженные**, указывая, что они были вставлены транзакцией, **зафиксированной максимально в прошлом**, так что эти данные с точки зрения MVCC определённо будут видны во всех текущих и будущих транзакциях.

Давайте для понимания схематично рассмотрим эту ситуацию (здесь использованы материалы статьи⁶⁸ на Хабр):

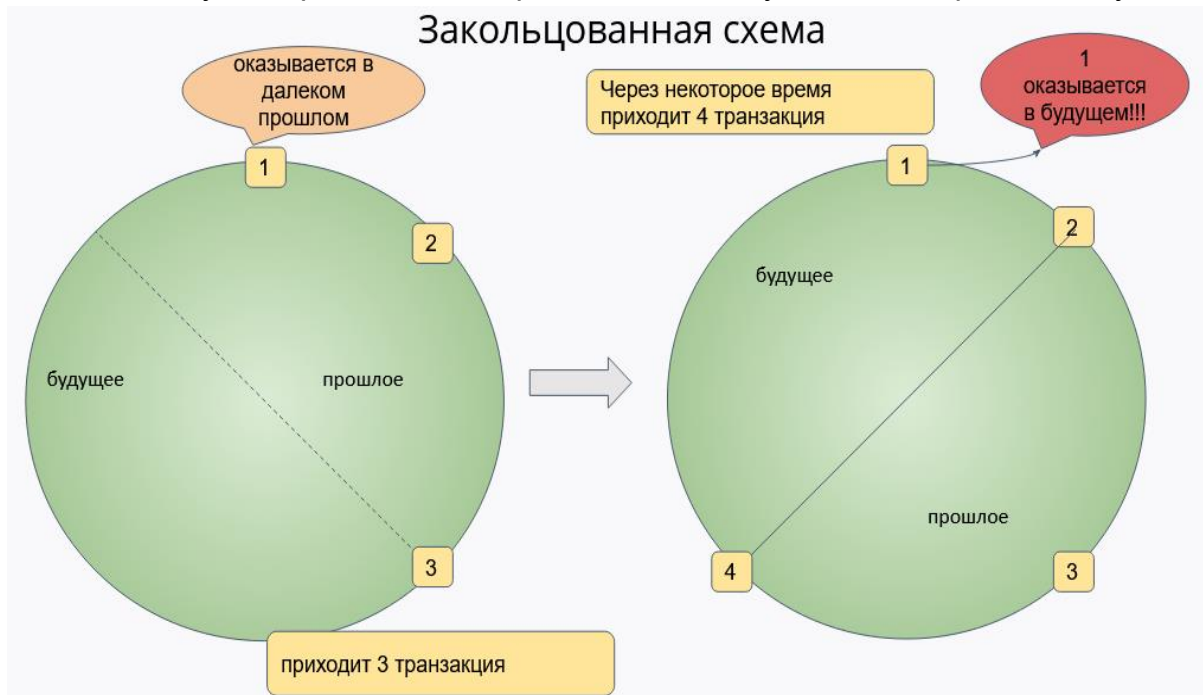


Итого, первая транзакция оказывается в прошлом.

⁶⁷ Freeze URL: <https://www.postgresql.org/docs/13/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND> (дата обращения 13.03.2021) [11]

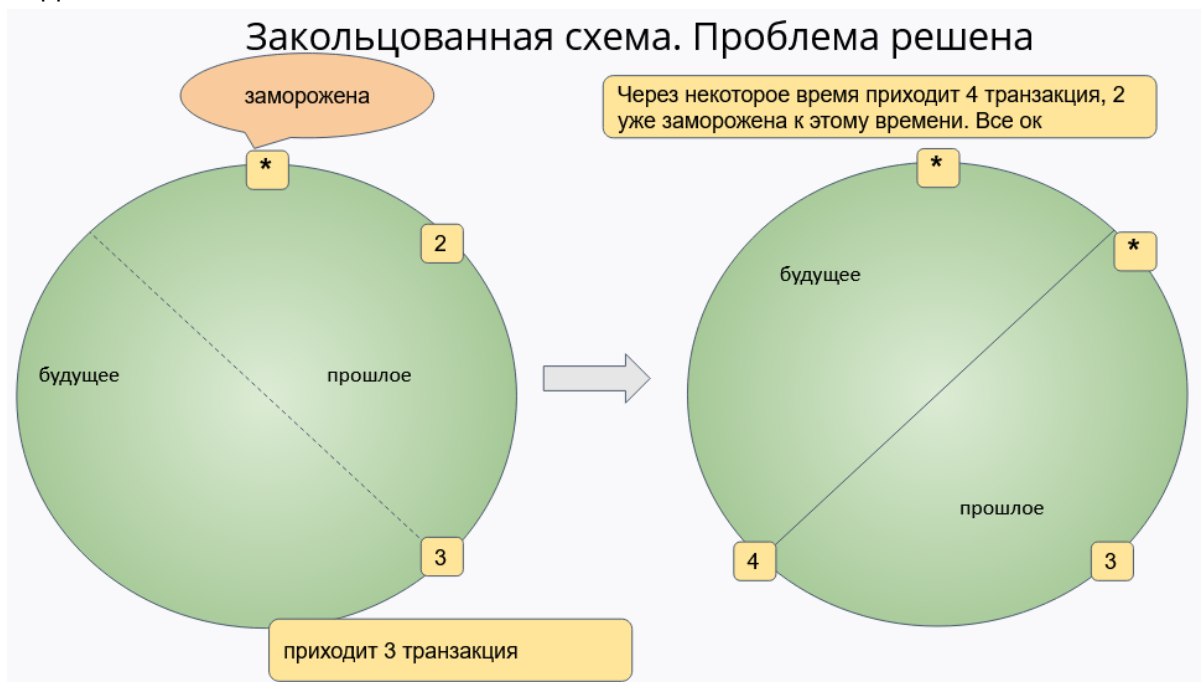
⁶⁸ MVCC в PostgreSQL-8. Заморозка URL: <https://habr.com/ru/company/postgrespro/blog/455590/> (дата обращения 13.03.2021) [12]

Затем у нас приходят 3 и 4 транзакции, и получается интересная ситуация:



Если транзакция в данном случае оказывается в будущем, то сервер аварийно завершит работу до устранения проблемы. Именно поэтому придуман механизм заморозки - при определённом пороге отставания от № текущей транзакции прошлые XID у записей замораживаются - т.е. у них проставляются биты **xmin_committed** и **xmin_aborted**.

Таким образом помечается, что эта запись считается очень старой и видной во всех снимках:



Теперь вернёмся к нашим параметрам вакуума и автовакуума:

vacuum_freeze_min_age - определяет, насколько старым должен стать XID, чтобы строки с таким XID были заморожены. Уменьшение приводит к увеличению количества транзакций, которые могут выполняться, прежде чем потребуется очередная очистка таблицы, а увеличение его значения помогает избежать ненужной работы, если строки, которые могли бы быть заморожены в ближайшее время, будут изменены ещё раз. Всё сугубо индивидуально на каждом проекте.

Обычная команда VACUUM не будет всегда замораживать все версии строк, имеющиеся в таблице, так как пропускает страницы, в которых нет мёртвых версий строк, даже если в них могут быть версии строк со старыми XID.

Периодически VACUUM будет также производить *агрессивную очистку*, пропуская только те страницы, которые не содержат ни мёртвых строк, ни незамороженных значений XID. Когда VACUUM будет делать это, зависит от параметра:

vacuum_freeze_table_age - полностью видимые, но не полностью замороженные страницы будут сканироваться, если число транзакций, прошедших со времени последнего такого сканирования, оказывается больше, чем **vacuum_freeze_table_age** минус **vacuum_freeze_min_age**. Если **vacuum_freeze_table_age** равно 0, VACUUM будет применять эту более агрессивную стратегию при каждом сканировании.

Максимальное время, в течение которого таблица может обходиться без очистки, составляет два миллиарда транзакций минус значение **vacuum_freeze_min_age** с момента последней агрессивной очистки.

Чтобы гарантировать, что это не произойдёт, для любой таблицы, которая может содержать значения XID старше, чем возраст, указанный в конфигурационном параметре **autovacuum_freeze_max_age**, вызывается автоочистка (Это случится, даже если автоочистка отключена).

Соответственно **autovacuum_freeze_max_age** - автоочистка будет вызываться приблизительно через каждые **autovacuum_freeze_max_age** минус **vacuum_freeze_min_age** транзакций. Работает только для таблиц, НЕ очищаемых регулярно.

Давайте на практике посмотрим на возраст самых старых данных в таблицах:

```
SELECT c.oid::regclass as table_name,  
       greatest(age(c.relFrozenxid),age(t.relFrozenxid)) as age  
FROM pg_class c  
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid  
WHERE c.relkind IN ('r', 'm');
```

table_name	age
test2	10
test	11
pg_statistic	27
pg_type	27
pg_foreign_table	27
pg_authid	27
pg_statistic_ext_data	27
pg_largeobject	27
pg_user_mapping	27
pg_subscription	27
pg_attribute	27
pg_proc	27
pg_class	27
pg_attrdef	27
pg_constraint	27
pg_inherits	27
pg_index	27
pg_operator	27
pg_opfamily	27
pg_opclass	27
pg_am	27
pg_amop	27
pg_amproc	27

Кластер практически пустой и возраст самых старых транзакций довольно молодой.

В этой главе мы с вами рассмотрели проблематику ACID и параллельных транзакций. Рассмотрели MVCC, как это реализовано в Постгресе и как за собой удалять мёртвые строки.

По умолчанию параметры автовакуума настроены достаточно хорошо. Менять их рекомендую при наличии каких-либо проблем, либо чётко понимая, зачем вы это делаете. Одна из самых распространённых рекомендаций от других специалистов - настраивать максимально агрессивно - имеет тоже место быть.

Таким образом мы закончили блок физического представления данных в СУБД. Переходим к уровням изоляции транзакций, логическому устройству Постгреса и внутренним механизмам реализации отказоустойчивости и реализации блокировок.

В конце главы, интересный кейс: как у Амазона была проблема с работой автовакуума и как они её решили⁶⁹.

Ссылка под №13 в соответствующем файле на гитхабе https://github.com/aeuge/Postgres13book_v2/blob/main/chapters/CHAPTER04.md

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/04/mvcc.txt

⁶⁹ Amazon тюнинг автовакуума URL: <https://aws.amazon.com/ru/blogs/database/a-case-study-of-tuning-autovacuum-in-amazon-rds-for-postgresql/> (дата обращения 13.03.2021) [13]

5. Уровни изоляции транзакций

В главе про MVCC мы обсуждали, для чего нужны транзакции. Но не всё так просто при конкурентном доступе.

В стандарте SQL есть четыре уровня изоляции транзакций⁷⁰, которые определяются в терминах аномалий⁷¹, которые допускаются при конкурентном выполнении транзакций на этом уровне:

- «Грязное» чтение (**Dirty read**) - транзакция T1 может читать строки, изменённые, но ещё не зафиксированные, транзакцией T2 (не было COMMIT). Отмена изменений (ROLLBACK) в T2 приведёт к тому, что T1 прочтёт данные, которых никогда не существовало

- Неповторяющееся чтение (**Non-repeatable read**) - после того, как транзакция T1 прочтёт строку, транзакция T2 изменила или удалила эту строку и зафиксировала изменения (COMMIT). При повторном чтении этой же строки транзакция T1 видит, что строка изменена или удалена

- Фантомное чтение (**Phantom read**) - транзакция T1 прочтёт набор строк по некоторому условию. Затем транзакция T2 добавила строки, также удовлетворяющие этому условию. Если транзакция T1 повторит запрос, она получит другую выборку строк

- Аномалия сериализации (**Serialization anomaly**) - результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди

Схематично можно представить это в виде таблицы, где в строках перечислены уровни изоляции, а в столбцах возможные аномалии (да - возможно, - - невозможно, PG - Постгрес):

	«грязное» чтение	неповторяю- щееся чтение	фантомное чтение	аномалия сериализации
Read Uncommitted	Допускается, но не в PG	да	да	да
Read Committed	-	да	да	да
Repeatable Read	-	-	Допускается, но не в PG	да
Serializable	-	-	-	-

Давайте разбирать каждый уровень подробно.

⁷⁰ Уровни изоляции транзакций URL: <https://www.postgresql.org/docs/13/transaction-iso.html> (дата обращения 15.03.2021) [1]

⁷¹ Аномалии при конкурентных транзакциях URL: <https://itnan.ru/post.php?c=1&p=317884> (дата обращения 15.03.2021) [2]

Уровень изоляции Read Uncommitted.

В Постгресе не допускается.

Уровень изоляции Read Committed.

Уровень изоляции транзакции, выбираемый в Постгресе **по умолчанию**. В транзакции, работающей на этом уровне, запрос SELECT видит только те данные, которые были зафиксированы до начала транзакции или в параллельных транзакциях во время выполнения текущей. Конечно результаты изменений, внесённых ранее в этой же транзакции так же будут видны.

Он никогда не увидит незафиксированных изменений, внесённых в процессе выполнения запроса параллельными транзакциями.

Два последовательных оператора SELECT могут видеть разные данные даже в рамках одной транзакции, если какие-то другие транзакции зафиксируют изменения после запуска первого SELECT, но до запуска второго.

Посмотрим на практике.

Подключимся к БД App:

\c app

Создадим таблицу testA:

CREATE TABLE testA (i serial, amount int);

Вставим два значения:

INSERT INTO testA(amount) VALUES (100), (500);

Посмотрим, что получилось:

SELECT * FROM testA;

```
app=# \c app
You are now connected to database "app" as user "postgres".
app=# CREATE TABLE testA (i serial, amount int);
CREATE TABLE
app=# INSERT INTO testA(amount) VALUES (100), (500);
INSERT 0 2
app=# select * from testA;
 i | amount 
---+-----
 1 |    100
 2 |    500
(2 rows)
```

Обратим внимание на то, что значения в поле *i* сгенерировались автоматически. Это произошло из-за того, что мы объявили тип поля **serial**, а это сокращённая запись для **integer NOT NULL DEFAULT nextval('имя_таблицы_имя_столбца_seq')**.

Давайте подробно посмотрим на таблицу testA и автоматически созданную последовательность:

\d+ testA

\d+ testa_i_seq

```
app=# \d+ testA
```

Table "public.testa"							
Column	Type	Collation	Nullable	Default	Storage	Stats target	Description
i	integer		not null	nextval('testa_i_seq'::regclass)	plain		
amount	integer				plain		

Access method: heap

```
app=# \d+ testa_i_seq
```

Sequence "public.testa_i_seq"						
Type	Start	Minimum	Maximum	Increment	Cycles?	Cache
integer	1	1	2147483647	1	no	1

Owned by: public.testa.i

Обратите внимание, что несмотря на то, что мы создавали таблицу testA в разном регистре, по умолчанию всё свелось к нижнему регистру.

Можно работать с объектом в разном регистре, использовать пробелы в имени или русский язык в названии таблиц/полей - при этом необходимо заключать такие названия в кавычки.

НО!!! Это НЕ рекомендованный способ именования.

Посмотрим на текущий параметр сессии AUTOCOMMIT:

\echo :AUTOCOMMIT - он говорит о том, что после каждого ввода команды в psql у нас происходит автоматическое оборачивание этой команды в транзакцию, то есть при включенном автокоммите просто запрос **SELECT** будет по факту выглядеть так:

BEGIN TRANSACTION;
SELECT ...
COMMIT;

Отключим автокоммит, так как нам необходимо посмотреть, что видят транзакции внутри себя при изменении данных в параллельной транзакции:

\set AUTOCOMMIT OFF

Посмотрим текущий уровень изоляции транзакций:

SHOW TRANSACTION ISOLATION LEVEL;

```
app=# \echo :AUTOCOMMIT
on
app=# \set AUTOCOMMIT OFF
app=# SHOW TRANSACTION ISOLATION LEVEL;
transaction_isolation
-----
read committed
(1 row)
```

Если нужно установить как раз уровень repeatable read можем выполнить команду **SET TRANSACTION ISOLATION LEVEL READ COMMITTED**;

Таким образом устанавливаем уровень изоляции на всю сессию. Если нам для какой-то конкретной транзакции нужен конкретный уровень транзакции, мы его просто указываем как параметр **ISOLATION LEVEL** при старте транзакции. Например, **BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE**;

После того, как отключили автокоммит, при вводе следующей команды автоматически начнётся транзакция на установленном уровне изоляции для нашей сессии (**при создании всегда read committed**).

Чтобы чётко понимать, когда она началась, рекомендую прямо так и писать - **BEGIN TRANSACTION**; или просто **BEGIN**; или **START TRANSACTION**; - аналогичные команды.

Посмотрим на практике, что произойдёт в транзакции, если её начать и в ней произойдёт ошибка:

```
app=# begin;
BEGIN
app=# select * fr testA;
ERROR:  syntax error at or near "fr"
LINE 1: select * fr testA;
              ^
app=!# select * from testA;
ERROR:  current transaction is aborted, commands ignored until end of transaction block
app=!# select * from testA;
ERROR:  current transaction is aborted, commands ignored until end of transaction block
app=!# rollback;
ROLLBACK
```

Любая команда после ошибочной выполнена не будет и нам в любом случае придётся откатить транзакцию - **ROLLBACK**;

Создадим второе подключение к нашей БД в другой консоли и отключим автокоммит:

\set AUTOCOMMIT OFF

В первой консоли начинаем транзакцию и смотрим, что в нашей таблице всего две записи, затем во второй консоли начинаем транзакцию и изменяем данные в этой таблице, и фиксируем изменения (**COMMIT**):

```
app=# begin;
BEGIN
app=# update testA set amount = 555 where i = 1;
UPDATE 1
app=# commit;
```

Видим, что если ещё раз в первой консоли в той же транзакции запросить данные, то они изменятся. При этом ещё и порядок изменится, так как селект читает данные подряд, а у нас, в связи с мультиверсионностью, строка дописывается физически в свободное место, в данном случае, в конец:

```
app=# select * from testA;
 i | amount 
---+-----
 1 |    100
 2 |    500
(2 rows)

app=# select * from testA;
 i | amount 
---+-----
 2 |    500
 1 |    555
(2 rows)
```

Таким образом, нужно понимать, что когда мы проектируем систему с данным уровнем изоляции, возможны такие разногласия в рамках одной транзакции. Давайте рассмотрим более строгий уровень изоляции.

Уровень изоляции Repeatable Read.

В этом режиме видны только те данные, которые были зафиксированы до начала транзакции и предыдущие изменения в своей транзакции.

В Постгрес этот уровень чуть усилен, не допускается даже фантомное чтение, за исключением аномалий сериализации.

Другими словами, этот уровень **отличается от Read Committed** тем, что запрос в транзакции данного уровня видит снимок данных на момент начала первого оператора в транзакции.

Таким образом, последовательные команды SELECT в **одной** транзакции **видят** одни и те же данные; они **не видят** изменений, внесённых и **зафиксированных другими** транзакциями **после начала** их текущей транзакции.

Посмотрим на практике.

Установим нужный уровень изоляции и начнём транзакции в двух сессиях.

В первой консоли:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SELECT * FROM testA;
```

Во второй добавим новую строку и закоммитим изменения:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
INSERT INTO testA VALUES (777);  
COMMIT;
```

```
app=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SET  
app=# INSERT INTO testA VALUES (777);  
INSERT 0 1  
app=# COMMIT;  
COMMIT  
app=# |
```

Посмотрим, что изменилось в первой сессии:

```
app=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SET  
app=# BEGIN;  
WARNING: there is already a transaction in progress  
BEGIN  
app=# SELECT * FROM testA;  
 i | amount  
---+-----  
 2 |    500  
 1 |    555  
(2 rows)  
  
app=# SELECT * FROM testA;  
 i | amount  
---+-----  
 2 |    500  
 1 |    555  
(2 rows)
```

Здесь обратим внимание на два момента:

- мы по-прежнему не видим аномалии “фантомное чтение” - новая строка не появилась в первой транзакции - но это только в Постгресе, в остальных СУБД она бы появилась
- мы попытались начать транзакцию командой BEGIN, и нам было сказано, что транзакция уже начата - она начинается автоматически после любой DDL-команды. Точнее, DDL вызывает rollback предыдущей транзакции и начинает новую. Обращайте на это внимание

Уровень изоляции **Serializable**⁷².

Этот уровень обеспечивает самую строгую изоляцию транзакций. На этом уровне моделируется последовательное выполнение всех зафиксированных транзакций, как если бы транзакции выполнялись одна за другой, последовательно, а не параллельно.

Фактически этот режим изоляции работает так же, как и Repeatable Read, только он дополнительно отслеживает условия, при которых результат параллельно выполняемых сериализуемых транзакций может не согласовываться с результатом этих же транзакций, выполняемых по очереди.

Соответственно на этом уровне приложения должны быть готовы повторять транзакции из-за сбоев сериализации.

Так как нам нужно отслеживать все изменения в параллельных транзакциях этот уровень создаёт некоторую добавочную нагрузку связанную с блокировкам.

При выявлении исключительных условий регистрируется аномалия сериализации и происходит сбой сериализации и происходит rollback транзакции.

Посмотрим в теории.

Сымитируем следующую ситуацию - у нас есть таблица с классами и значениями:

class		value
-----+-----		
1		10
1		20
2		100
2		200

Первая сериализуемая транзакция А вычисляет:

SELECT SUM(value) FROM testS WHERE class = 1;

и добавляет запись с суммой и class 2 в эту таблицу, и в теории получаем:

class		value
-----+-----		
1		10
1		20
2		100
2		200
2		230

⁷² Аномалии при уровне изоляции транзакций Serializable URL: <https://www.postgresql.org/docs/13/transaction-iso.html#XACT-SERIALIZABLE> (дата обращения 24.09.2021) [3]

Вторая сериализуемая транзакция В вычисляет:

SELECT SUM(value) FROM testS WHERE class = 2;

И добавляет запись с суммой и class 1 в эту таблицу, и в теории получаем:

class	value
1	10
1	20
2	100
2	200
1	300

Вопрос - что произойдёт при попытке фиксации изменений?

Если бы одна из этих транзакций закоммитилась раньше другой - то значения были бы другие, а так как на уровне **SERIALIZABLE** движок пытается выстроить все транзакции последовательно, как будто бы не было параллельных транзакций, у нас ничего не получится.

Если первая выполнится раньше - действия второй будут неверны и наоборот.

На практике.

Для этого создадим таблицу для тестов:

CREATE TABLE testS (i int, amount int);

Вставим записи как в примере:

INSERT INTO TESTS VALUES (1,10), (1,20), (2,100), (2,200);

В первой сессии устанавливаем нужный уровень и начинаем транзакцию во втором окне, и имитируем SELECT и INSERT, и получаем следующую картину:

В первой сессии:

```
app=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
app=# SELECT sum(amount) FROM testS WHERE i = 1;
sum
-----
 30
(1 row)

app=# SELECT sum(amount) FROM testS WHERE i = 1;
sum
-----
 30
(1 row)
```

Во второй сессии:

```
app=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
app=# SELECT sum(amount) FROM testS WHERE i = 2;
 sum
-----
 300
(1 row)

app=# INSERT into testS VALUES (1,200);
INSERT 0 1
```

А теперь самое интересное - коммит в первой сессии пройдет нормально, а во второй нас ждёт сюрприз:

```
app=# COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Рекомендации при применении уровня **SERIALIZABLE**:

- Заключайте в одну транзакцию не больше команд, чем необходимо для обеспечения целостности
- Не оставляйте соединения «простаивающими в транзакции» дольше, чем необходимо. Для автоматического отката долгих транзакций можно изменить параметр конфигурации **idle_in_transaction_session_timeout**
- Объявляйте транзакции как READ ONLY, если это отражает их суть
- Управляйте числом активных подключений, при необходимости используя пул соединений. Это всегда полезно для увеличения производительности, но особенно важно в загруженной системе с сериализуемыми транзакциями
- Последовательное сканирование всегда влечёт за собой предикатную блокировку (рассмотрим в главе про блокировки). Это приводит к увеличению сбоев сериализации. В таких ситуациях может помочь склонение оптимизатора к использованию индексов, уменьшая **random_page_cost** и/или увеличивая **cpu_tuple_cost**. Главное сопоставить выигрыш от уменьшения числа откатов и перезапусков транзакций с проигрышем от возможного менее эффективного выполнения запросов

В данной главе рассмотрели, как работают конкурентные транзакции и какие при этом могут возникать различные аномалии, а также что с этим делать. В следующей главе будем рассматривать логическое устройство Постгреса.

В качестве бонуса почитайте статью про теоретико-практическое исследование уровня изоляции SERIALIZABLE⁷³ и дальнейшее предложение по улучшению производительности и решению проблем, связанных с сериализацией - двухфазный коммит⁷⁴. Забегая вперёд (не в рамках этой книги), сейчас есть более продвинутые варианты двухфазного коммита. Всё зависит от решаемых задач и области применения.

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/Postgres13book_v2/blob/main/scripts/05/isolation.txt

⁷³ Теория про сериализацию URL: <https://arxiv.org/pdf/1208.4179.pdf> (дата обращения 15.03.2021) [4]

⁷⁴ Двухфазный коммит URL: https://en.wikipedia.org/wiki/Two-phase_commit_protocol (дата обращения 15.03.2021) [5]