PostgreSQL 18:
для архитекторов систем.

УДК 004.65
ББК 16.262
А81

А81 Аристов Евгений. PostgreSQL 18: для архитекторов систем. – М.: ООО «Сам Полиграфист», 2026. – 520 с.

ISBN 978-5-00227-766-7

В данной книге подробно изучается архитектура PostgreSQL 18, начиная с физического устройства этой системы управления базами данных и заканчивая репликацией и оптимизацией работы.

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 устроен PostgreSQL изнутри. На практике показана архитектура работы с данными, особенности реализации механизмов MVCC, TOAST, VACUUM, LOCK и многих других. Объяснено, почему настройка параметров подключения к PostgreSQL важна и обязательна. Подробно рассматриваются настройки, отвечающие за производительность, безопасность и варианты их оптимизации в зависимости от разных факторов. Впервые разбирается вариант бесшовного секционирования практически без даунтайма и дополнительного места.

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

Книга будет интересна и полезна широкой аудитории: разработчикам, администраторам баз данных, девопс-инженерам, архитекторам программного обеспечения.

Тираж 100 экз. Заказ № 67350.

Отпечатано в типографии «OneBook.ru»
ООО «Сам Полиграфист»
129090 г. Москва, Волгоградский проспект, д. 42, корп. 5
www.onebook.ru

© Аристов Е.Н., 2026.

Оглавление

Об авторе

Для начала, хотелось бы поблагодарить за то, что вы выбрали эту книгу. Я в IT уже больше 25 лет. Начинал, как и многие когда-то, с ZX Spectrum. Написал свою первую игру наподобие текстовой Dictator. Тогда всё 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 успешно реализованных проектов с применением виртуализации, кластеризации и highload. В какой-то момент понял, что хочу учить людей, нести светлое и доброе в наш мир. Уже больше шести лет я читаю свои авторские курсы по PostgreSQL и другим направлениям во множестве обучающих организаций, институтов и университетов, пишу книги по базам данных.

За это время издал несколько книг по практическому разбору архитектуры, встречаемых проблемах, методам их решения и лучшим рецептам по оптимизации производительности самой популярной СУБД[1] – PostgreSQL. В этой книге даются практические рекомендации, проведён глубокий анализ внутреннего устройства для понимания принципов работы этой СУБД и возможностей её тонкой настройки. Рассматриваются частые ошибки при проектировании и использовании PostgreSQL, а также лучшие варианты тюнинга (тонкой настройки).

В 2021 году создал личный сайт-визитку https://aristov.tech, где были только регалии и контакты. Сейчас сайт вырос до обучающего портала, на нём доступны:

- блог с интересными статьями
- менторинг для ускоренного обучения почти по всем направлениям
- каналы YouTube, Rutube, VKvideo с обучающими роликами
- доступны два обучающих курса: по основам SQL и продвинутый по серверному программированию – PL/pgSQL
- доступны к заказу практические воркшопы по устройству PostgreSQL (wal, секционирование и другие)
- сложные практические воркшопы по Patroni (разбираем георазпределённую мультидатацентровую конфигурацию, устраняем аварии, обсуждаем тонкости сетевых настроек и тестируем отказоустойчивость)
- для B2B доступна линейка курсов для обучения сотрудников
- вишенкой на торте является мой уникальный курс по оптимизации PostgreSQL

[1] система управления базами данных

Эта книга написана под новую версию PostgreSQL – 18. Однако большинство из используемых подходов к решению проблем подойдут как к более старым версиям PostgreSQL, так и, что более важно, к новым версиям!

Ссылки из данной книги расположены на GitHub[2] для удобства перехода в бумажном варианте. В репозитории ссылки будут разбиты по номерам тем, номер ссылки в квадратных скобках [1]. Также на сайте https://aristov.tech можно заказать[3] оригинальную электронную версию в PDF или бумажную версию с автографом. Также можно заказать предыдущую книгу по PostgreSQL 16: лучшие практики оптимизации. – М.: ООО «Сам Полиграфист», 2024. – 316 с. Отрывок из неё или текущей книги всегда можно найти на моём сайте и принять решение о заказе. На сайте бесплатно доступна книга в PDF по развёртыванию отказоустойчивых решений PostgreSQL – PostgreSQL 14. Оптимизация, Kubernetes, кластеры, облака. – М.: ООО «Сам Полиграфист», 2022. – 576 с.

Спасибо Алексею Цыкунову @erlong15. Мой наставник, Профессионал с большой буквы, опытнейший DBA[4] и просто замечательный человек. Сооснователь Гилберттим (hilbertteam.com). Спасибо Дмитрию Буну @dmitriyboon за неоднократные возможности участвовать в крутых архитектурных проектах и фантастическую обложку к этой книге. Спасибо любимой жене Светлане @asveta за поддержку. Спасибо редактору книги Андрею Alandre @alandreei. Ну и, конечно же, семье, которая меня поддерживала все эти долгие годы авторского труда.

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

[2] мой GitHub URL: https://github.com/aeuge/postgres18book (дата обращения 16.03.2026) [0]
[3] ссылка для заказа книги URL: https://aristov.tech/#orderbook (дата обращения 16.03.2026) [1]
[4] DataBase Architector/Administrator

1. Основы PostgreSQL

Начну книгу с новостей о последнем, на дату написания, релизе – версия 18 вышла 15.09.2025. Основная презентация релиза[5], изменения и дополнения в PostgreSQL 18 доступны по ссылке[6].

Стоит отметить основные, с моей точки зрения, изменения:

- добавлена подсистема асинхронного ввода/вывода[7], позволяющая увеличить пропускную способность ввода/вывода и избавиться от задержек. В Linux добавили интерфейс асинхронного ввода/вывода io_uring[8] (**io_method=io_uring**), поддерживаемый начиная с ядра Linux 5.1[9]. Теперь и PostgreSQL может задействовать асинхронный ввод/вывод на новых условиях, пока только для ускорения выполнения некоторых операций, связанных с чтением данных из файловой системы, таких как последовательный перебор, сканирование битовой карты индексов и проведение чистки (vacuum). В *некоторых тестах* применение AIO приводит к увеличению производительности в два-три раза
  - *а на стандартных тестах разницы не обнаружил, подождём, возможно покажут методику или ситуации, когда будет прирост производительности*
- добавлены оптимизации, более эффективно использующие индексы для запросов, содержащих конструкции «OR» и «IN (...)» в блоке «WHERE», а также повышающие производительность планирования и выполнения объединения таблиц (например, ускорен код слияния хэшей и разрешено использовать инкрементальную сортировку при слиянии таблиц)
  - *в бизнесовых реалиях какого-то ощутимого ускорения не увидел*
- добавлена поддержка виртуальных генерируемых столбцов, значение которых вычисляется на лету в процессе выполнения запросов, без сохранения на диск – **GENERATED VIRTUAL**) – с версии 12, когда они были представлены, была возможность их только хранить на диске (**STORED**)

---

[5] релиз PostgreSQL URL: https://www.postgresql.org/about/news/postgresql-18-released-3142/ (дата обращения 16.03.2026) [2]
[6] изменения в релизе URL: https://www.postgresql.org/docs/17/static/release-18.html (дата обращения 16.03.2026) [3]
[7] асинхронный ввод вывод URL: https://wiki.postgresql.org/wiki/AIO (дата обращения 16.03.2026) [4]
[8] io_uring URL: http://kernel.dk/io_uring.pdf (дата обращения 16.03.2026) [5]
[9] ядро Linux 5.1 URL: https://www.opennet.ru/opennews/art.shtml?num=50631 (дата обращения 16.03.2026) [6]

- ➢ *по факту это замена ручных хранимых функций, выполняющих аналогичный функционал – синтаксический сахар. Крайне редко используемый функционал*

❖ в командах INSERT, UPDATE, DELETE и MERGE реализована возможность вывода прошлых (OLD) и текущих (CURRENT) значений в выражении RETURNING. Например: «UPDATE... RETURNING WITH (OLD AS o, NEW AS n) o.*, n.*»

- ➢ *интересно – но зачем? новые значения и так знаем у апдейта, прошлого у INSERT и так нет, как и нового у DELETE*

❖ добавлена функция **uuidv7()**[10] для генерации случайных уникальных идентификаторов в формате UUIDv7[11]

- ➢ В отличие от старой функции для генерации UUID (gen_random_uuid), которая теперь дополнительно доступна под именем uuidv4(), в UUIDv7 помимо случайного значения включается время генерации. Наличие упорядоченных частей в значении UUID (первые 12 символов — эпохальное время, а последующие 18 — случайное значение) повышает эффективность сортировки и индексирования, что актуально, так как UUID обычно используются для первичных ключей (например, ключи, созданные в близкое время размещаются рядом друг с другом в индексе)
  - ■ *хорошее нововведение! Ждали 3-4 года!*
  - ■ *пример ускорения работы – снижения нагрузки на диск в два раза в компании Тензор*[12]

❖ в новых установках **включено по умолчанию** использование контрольных сумм для проверки целостности хранимых данных. Для отмены данного поведения при запуске initdb следует указать опцию --no-data-checksums

- ➢ *ожидаемо*

❖ в версии 18 в pg_dump можно включать **статистику**, чтобы не пересчитывать её при восстановлении

- ➢ *очень удачное решение*

❖ в версию 17.6 включили новую опцию в pg_dump, но в релизе версии 18 про это ни слова (мало кто читает минорные релизы и в любом случае можно было пропустить эту новость) – **\restrict**-режим – после него игнорируются любые команды, раньше злоумышленники могли встроить зловредный код для выполнения при восстановлении, сейчас

---

[10] uuid PostgreSQL URL: https://www.postgresql.org/docs/18/functions-uuid.html (дата обращения 16.03.2026) [7]
[11] uuid v7 URL: https://uuidv7.org/ (дата обращения 16.03.2026) [8]
[12] статья на Хабр про uuid v7 URL: https://habr.com/ru/companies/tensor/articles/989032/ (дата обращения 16.03.2026) [9]

будет выполнена только следующая команда \unrestrict и hash-код команды запрета

- ➢ *давно следовало так сделать*

- ❖ остальное классический косметический тюнинг

Далее необходимо рассмотреть стенды, на которых можно практиковаться. В зависимости от аппаратных возможностей, ОС[13] и версии PostgreSQL, существует несколько вариантов. Они подробно рассмотрены в первой главе моей книги по PostgreSQL 14[14] и в статьях в блоге, поэтому останавливаться подробно я на них не буду. Создание ВМ в одном из трёх вариантов VirtualBox[15], ЯндексОблако[16], ГуглОблако[17].

Выбирайте тот вариант, который проще. Рекомендую или локальный VirtualBox (доступен на любой ОС, главное, чтобы хватило ресурсов), или использовать ЯндексОблако[18] (доступен приветственный бонус, а если выключать ВМ на время, когда она не нужна, то стоимость довольно демократична).

Если взять классический инстанс для обучения: два ядра, четыре гигабайта оперативной памяти и диск на 20 гигабайт, за основу взять ОС Ubuntu[19] (рекомендую LTS[20] 24.04 или уже после отправки книги на печать выйдет 26.04 версии – стабильные версии с пятилетней поддержкой) – то это будет стоить ~3000 рублей в месяц, 100 рублей в день или 4 рубля в час. В день 2-3 часа на изучение PostgreSQL на основе моих примеров, думаю, не сильно ударит по вашему бюджету.

Главное установить конкретную версию PostgreSQL 18, но в целом 99% примеров будут работать и на других версиях. Простой вариант установки PostgreSQL рассмотрен в статье[21], более продвинутые варианты перечислены в первой главе моей книги по PostgreSQL 14.

Классическая команда для обновления списка пакетов, добавления репозитория и установки PostgreSQL 18:

---

[13] операционной системы
[14] PostgreSQL 14. Оптимизация, Kubernetes, кластера, облака. – М.: ООО «Сам Полиграфист», 2022. – 576 с.
[15] развёртывание виртуальных машин в Virtualbox URL: https://aristov.tech/blog/prostoj-sposob-razvernut-virtualnuyu-mashinu-na-prakticheski-lyubom-komputere-noutbuke-ispolzuya-virtualbox/ (дата обращения 16.03.2026) [10]
[16] развёртнывание виртуалььных машин в ЯндексОблаке URL: https://aristov.tech/blog/deploj-vm-v-yandeks-oblake/ (дата обращения 16.03.2026) [11]
[17] развёртнывание виртуальных машин в ГуглОблаке URL: https://aristov.tech/blog/dva-prostyh-sposoba-razvernut-virtualnuyu-mashinu-v-prostranstve-google-compute-engine/ (дата обращения 16.03.2026) [12]
[18] ЯндексОблако URL: https://yandex.cloud/ru (дата обращения 16.03.2026) [13]
[19] Ubuntu URL: https://ubuntu.com/ (дата обращения 16.03.2026) [14]
[20] релизный цикл Ubuntu URL: https://ubuntu.com/about/release-cycle (дата обращения 16.03.2026) [15]
[21] варианты установки PostgreSQL URL: https://aristov.tech/blog/varianty-ustanovki-postgresql/ (дата обращения 16.03.2026) [16]

```
sudo apt update && sudo DEBIAN_FRONTEND=noninteractive 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 DEBIAN_FRONTEND=noninteractive apt -y install postgresql-18
```

Если вам необходим доступ к установленной версии по сети, а не с локальной консоли (только туда по умолчанию и есть доступ), необходимо настроить подключение к PostgreSQL. Подробное описание доступно в блоге – из командной строки[22] или GUI для работы[23].

Вкратце, необходимо три основных шага:

Первое — перейдите в консоли Linux под Linux-пользователя postgres и задайте **сложный** пароль пользователю СУБД postgres (пользователи postgres в Linux и PostgreSQL это разные пользователи) через утилиту, входящую в поставку PostgreSQL – psql[24] (подробно, как ей пользоваться, рассказано во второй главе моей книги по PostgreSQL 14):

```
sudo su postgres
ALTER USER postgres PASSWORD 'Postgres123#';
```

или используя встроенную команду psql (начинаются с обратного слеша):

```
\password
```

Второе — включите listener (процесс, слушающий подключения на сетевом интерфейсе) в postgresql.conf, раскомментируйте соответствующую строку, убрав символ # в текстовом редакторе nano[25]:

```
sudo nano /etc/postgresql/18/main/postgresql.conf
listen_addresses = ''  # IP адреса, на которых PostgreSQL принимает сетевые подключения, например, localhost, 10… или * для доступа из любой доступной сети – не рекомендовано.
```

Обратите внимание, кавычки должны быть прямые, а открывать доступ в интернет стоит осознанно и очень внимательно. Иначе злоумышленники могут подобрать пароль и воспользоваться вашим инстансом для своих целей.

---

[22] подключение из командной строки URL: https://aristov.tech/blog/podklyuchenie-k-postgresu-iz-komandnoj-stroki/ (дата обращения 16.03.2026) [17]

[23] подключение из графической оболочки URL: https://aristov.tech/blog/podklyuchenie-k-postgresu-iz-gui/ (дата обращения 16.03.2026) [18]

[24] psql URL: https://www.postgresql.org/docs/current/app-psql.html (дата обращения 16.03.2026) [19]

[25] nano URL: https://www.nano-editor.org/ (дата обращения 16.03.2026) [20]

Третье — укажите вход по паролю в pg_hba.conf и измените маску подсети, откуда будет разрешён доступ к нашему кластеру.

```
sudo nano /etc/postgresql/18/main/pg_hba.conf
host all all 0.0.0.0/0 scram-sha-256
```

Обязательно укажите метод шифрования пароля:

- **scram-sha-256** — современный метод шифрования, доступен с версии 18
- **md5** — устарел с выходом версии 18. Передаётся в зашифрованном виде. Используется для совместимости со старыми клиентами. Не рекомендован, так как недостаточно защищён в современных реалиях. Например, существуют радужные таблицы[26], с помощью которых злоумышленник, получив хэш пароля[27] (однонаправленное шифрование), может получить его в открытом виде
- **password** — пароль будет передан в открытом виде и может быть перехвачен злоумышленником. Не рекомендовано!

Обратите внимание, возможно придётся написать правило для облачного фаервола для доступа извне облака.

После этого перезагрузите кластер PostgreSQL и получите доступ извне:

```
pg_ctlcluster 18 main stop && pg_ctlcluster 18 main start
```

Обратите внимание, просто restart под Linux-пользователем postgres скорее всего не сработает, так как экземпляр после установки запущен под суперпользователем. Или нужно под пользователем повысить права до супер через sudo[28] и перезагрузить кластер.

Итого, получили работающий экземпляр кластера (в PostgreSQL принята такая терминология, что каждый инстанс называется кластером). Чтобы увидеть состояние кластера, можно воспользоваться Linux-командой **pg_lsclusters**[29] (входят в комплект поставки для Debian-похожих систем, Ubuntu входит в их число, для других ОС может потребоваться ручное управление через **pg_ctl**[30]):

---

[26] радужные таблицы URL: https://en.wikipedia.org/wiki/Rainbow_table (дата обращения 16.03.2026) [21]
[27] хэш URL: https://pasgen.ru/hash (дата обращения 16.03.2026) [22]
[28] sudo URL: https://www.sudo.ws/ (дата обращения 16.03.2026) [23]
[29] pg_lsclusters URL: https://manpages.ubuntu.com/manpages/jammy/man1/pg_lsclusters.1.html (дата обращения 16.03.2026) [24]
[30] pg_ctl URL: https://www.postgresql.org/docs/current/app-pg-ctl.html (дата обращения 16.03.2026) [25]

```
aeugene@Aeuge:~$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory              Log file
18  main    5432 online postgres /var/lib/postgresql/18/main /var/log/postgresql/postgresql-18-main.log
```

В книге, сейчас и далее, кроме явно генерируемых примеров, используются заранее созданные мной БД[31] по автобусным пассажирским перевозкам в Таиланде[32] для PostgreSQL. Схема данных относительно простая – есть города, автобусные станции, автобусы, маршруты перевозок, расписание и сами перевозки.

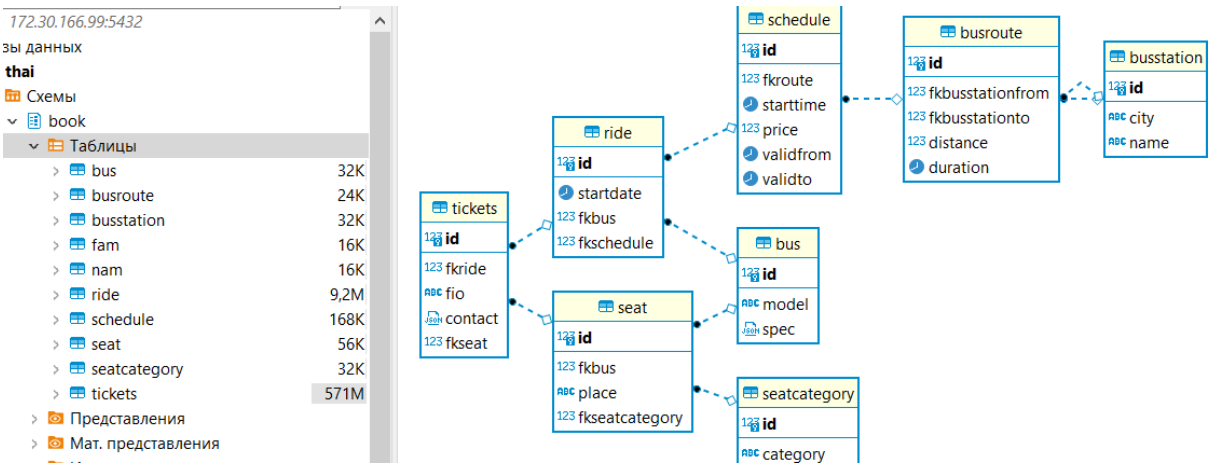
Всего предлагается три варианта: маленький, который будет в примерах, на ~6 млн строк (600 МБ[33]), средний на ~60 млн строк (6 ГБ[34]) и большой на ~600 млн строк (60 ГБ). Инструкции разворачивания остальных вариантов расположены на странице «Тайские перевозки» на GitHub.

Загрузить данные в инстанс PostgreSQL можно довольно легко утилитой wget[35], командой загрузки скачав заархивированный датасет с открытого Google Storage[36], его разархивирования утилитой tar[37] и перенаправления SQL-кода в PostgreSQL (знак перенаправления <), объединив всё через двойной амперсанд (&&) – под Linux-пользователем postgres:

```
sudo su postgres
cd ~ && wget https://storage.googleapis.com/thaibus/thai_small.tar.gz && tar -xf thai_small.tar.gz && psql < thai.sql
```

Схема данных загруженного датасета:

[31] База данных
[32] Тайские перевозки URL: https://github.com/aeuge/postgres16book/tree/main/database (дата обращения 16.03.2026) [26]
[33] Мегабайт
[34] Гигабайт
[35] wget URL: https://en.wikipedia.org/wiki/Wget (дата обращения 16.03.2026) [27]
[36] Облачное хранилище URL: https://cloud.google.com/storage?hl=ru (дата обращения 16.03.2026) [28]
[37] tar URL: https://en.wikipedia.org/wiki/Tar_(computing) (дата обращения 16.03.2026) [29]



Объём БД можно посмотреть, используя команду \l+ в утилите psql, она покажет объём загруженных данных. Для этого сначала укажем БД, к которой необходимо подключиться при запуске утилиты:

```
psql -d thai
\l+
```

```
thai      | postgres | UTF8     | C.UTF-8 | C.UTF-8 |                    | 590 MB  | pg_default |
```

Обратите внимание, воспользовались специальной командой psql, которая начинается с обратного слеша **** (они работают ТОЛЬКО в psql) – таких команд довольно много, чтобы можно было удобно пользоваться консольной командой, они предоставляют расширенный доступ к системным представлениям.

Итого, в этой главе развернули стенд и подготовили его к работе. В следующей главе рассмотрим, как открыть доступ к БД изнутри PostgreSQL.

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/01/intro.sql

2. Физическая архитектура

В предыдущей главе рассмотрели варианты выбора ВМ, установки PostgreSQL и настройки доступа извне.

Все команды из этой главы доступны в файле на GitHub https://github.com/aeuge/postgres18book/blob/main/scripts/02/physical.sql
Для каждой последующей главы есть свой соответствующий файл.

После установки посмотрим на список кластеров PostgreSQL через утилиту **pg_lsclusters** :

```
aeugene@Aeuge:~$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory              Log file
18  main    5432 online postgres /var/lib/postgresql/18/main /var/log/postgresql/postgresql-18-main.log
```

Видим, что пока развёрнут один кластер **PostgreSQL 18** по имени **main** на порту **5432**.

Кластер запущен и находится онлайн.

Владелец кластера – Linux-пользователь **postgres**.

Обратите внимание, что, несмотря на идентичное имя postgres *у суперпользователя PostgreSQL, это другой тип – пользователь ОС Linux.*

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.

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

```
aeugene@Aeuge:~$ pg_
pg_archivecleanup  pg_config          pg_dropcluster     pg_lsclusters      pg_renamecluster   pg_updatedicts
pg_backupcluster   pg_conftool        pg_dump            pg_receivewal      pg_restore         pg_upgradecluster
pg_basebackup      pg_createcluster   pg_dumpall         pg_receivexlog     pg_restorecluster  pg_virtualenv
pg_buildext        pg_ctlcluster      pg_isready         pg_recvlogical     pg_top
```

Рассмотрим основные из них:

- **pg_createcluster**[38] – создать кластер с нужными параметрами

[38] pg_createcluster URL: https://helpmanual.io/help/pg_createcluster/ (дата обращения 10.03.2026) [1]

- **pg_renamecluster**[39] – переименовать кластер
- **pg_dropcluster**[40] – удалить кластер
- **pg_ctlcluster**[41] – 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
- **pg_upgradecluster**[42] – обновить кластер на другую основную версию. На практике посмотрим в конце главы.

Например, добавим ещё один кластер под Linux-пользователем aeugene:
`pg_createcluster 18 main2`

```
aeugene@Aeuge:~$ pg_createcluster 18 main2
install: cannot change permissions of '/etc/postgresql/18/main2': No such file or directory
Error: could not create configuration directory; you might need to run this program with root privileges
```

Обратите внимание на следующие особенность – под нашим простым Linux-пользователем кластер создать не удалось, так как нет доступа к каталогу **/etc/postgresql/18**.

Тут есть три варианта:

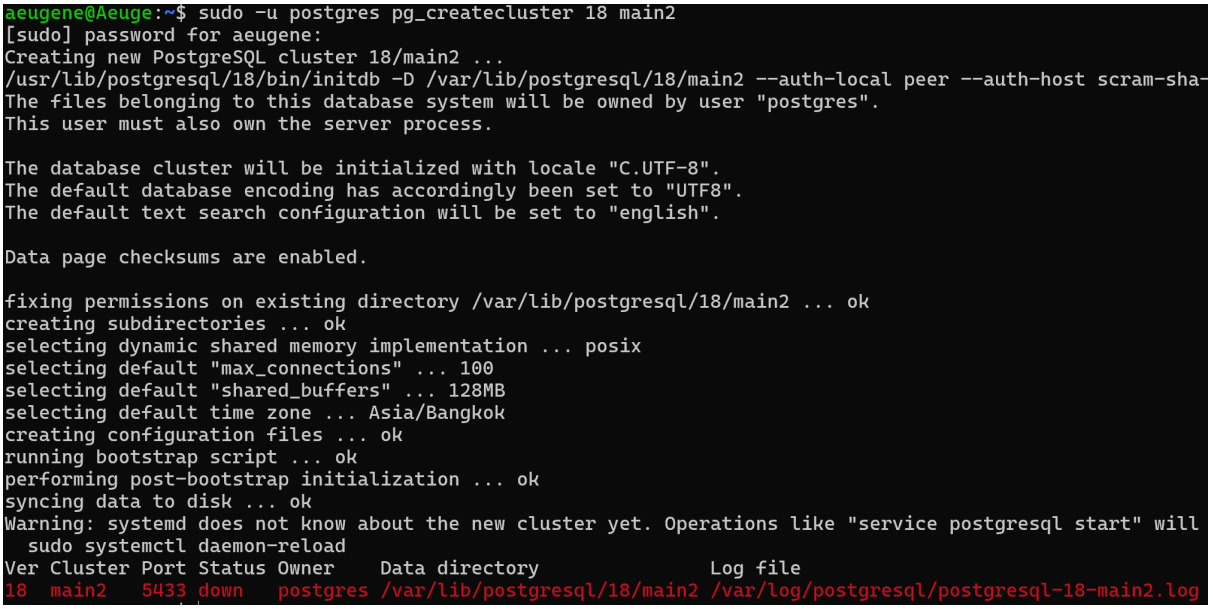
- переключиться на Linux-пользователя postgres и под ним создать кластер:
  - ➢ `sudo su postgres`
  - ➢ `pg_createcluster 18 main2`
- перейти под Linux-суперпользователя **root** командой и создать кластер (не рекомендую, в будущем, скорее всего, будут проблемы с правами на каталоги):
  - ➢ `sudo su`
  - ➢ `pg_createcluster 18 main2`
- использовать утилиту **sudo** для поднятия своих прав до прав суперпользователя и выполнить команду под Linux-пользователем postgres:
  - ➢ `sudo -u postgres pg_createcluster 18 main2`

---

[39] pg_renamecluster URL: https://www.onworks.net/programs/pg-renamecluster-online (дата обращения 10.03.2026) [2]
[40] pg_dropcluster URL: http://manpages.ubuntu.com/manpages/trusty/man8/pg_dropcluster.8.html (дата обращения 10.03.2026) [3]
[41] pg_ctlcluster URL: http://manpages.ubuntu.com/manpages/hirsute/en/man1/pg_ctlcluster.1.html (дата обращения 10.03.2026) [4]
[42] pg_upgradecluster URL: https://manpages.ubuntu.com/manpages/trusty/man8/pg_upgradecluster.8.html (дата обращения 10.03.2026) [5]

Добавим ещё один кластер, используя вариант с sudo:

```
sudo -u postgres pg_createcluster 18 main2
```



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 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/18/main/postgresql.conf**. После внесения изменений нажмите **Ctrl+X** и подтвердите изменения в файле, нажав кнопки **Y** и потом **ENTER**. Запустите кластер – кластер будет запущен уже на новом порту.

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

```
sudo su postgres
```

Перейдём в этот каталог и выполним команду:

```
cd /var/lib/postgresql/18/main
ls -la
```

```
postgres@Aeuge:~$ cd /var/lib/postgresql/18/main
postgres@Aeuge:~/18/main$ ls -la
total 92
drwx------ 19 postgres postgres 4096 Feb 26 13:59 .
drwxr-xr-x  4 postgres postgres 4096 Feb 26 15:05 ..
-rw-------  1 postgres postgres    3 Feb 25 15:07 PG_VERSION
drwx------  7 postgres postgres 4096 Feb 26 14:14 base
drwx------  2 postgres postgres 4096 Feb 26 14:15 global
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_commit_ts
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_dynshmem
drwx------  4 postgres postgres 4096 Feb 26 14:14 pg_logical
drwx------  4 postgres postgres 4096 Feb 25 15:07 pg_multixact
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_notify
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_replslot
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_serial
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_snapshots
drwx------  2 postgres postgres 4096 Feb 26 13:59 pg_stat
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_stat_tmp
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_subtrans
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_tblspc
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_twophase
drwx------  4 postgres postgres 4096 Feb 26 14:15 pg_wal
drwx------  2 postgres postgres 4096 Feb 25 15:07 pg_xact
-rw-------  1 postgres postgres   88 Feb 25 15:07 postgresql.auto.conf
-rw-------  1 postgres postgres  130 Feb 26 13:59 postmaster.opts
-rw-------  1 postgres postgres  107 Feb 26 13:59 postmaster.pid
```

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

В каталоге **global** находятся глобальные/системные объекты, которые присутствуют во всех БД кластера. В каталоге **base** расположены файлы с нашими базами данных. Но используются не имена баз данных, а их **OID – object id** (уникальный сквозной идентификатор всех объектов в кластере):

```
ls -l base
```

```
postgres@Aeuge:~/18/main$ ls -l base
total 28
drwx------ 2 postgres postgres  4096 Feb 26 14:02 1
drwx------ 2 postgres postgres 12288 Feb 26 14:15 16388
drwx------ 2 postgres postgres  4096 Feb 26 14:15 4
drwx------ 2 postgres postgres  4096 Feb 26 14:14 5
drwx------ 2 postgres postgres  4096 Feb 26 14:14 pgsql_tmp
```

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

```
psql
```

```
SELECT oid, datname, dattablespace FROM pg_database;
```

```
postgres=# SELECT oid, datname, dattablespace FROM pg_database;
  oid  |  datname  | dattablespace
-------+-----------+---------------
     5 | postgres  |          1663
 16388 | thai      |          1663
     1 | template1 |          1663
     4 | template0 |          1663
(4 rows)
```

Обратите внимание на разный знак приглашения в командной строке:
$ – находимся в консоли Linux
 # – находимся в консоли psql

На скриншоте 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 – не использовать
- но и не удалять – иногда нужна для различных утилит

Для своих целей создаём свою БД, используя **DDL**[43] **create database**[44]:

```
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 (ТП) 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 ТП по умолчанию, так и в каждом конкретном случае указывать, в каком ТП будет расположен конкретный объект (таблица, индекс, статистика и так далее) внутри базы данных
- ❖ один объект должен быть расположен только в одном ТП, но есть секционированные таблицы, для которых как раз можно использовать разные ТП

По умолчанию есть два ТП. 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 ТП по умолчанию создаются в каталоге **$PGDATA\pg_tblspc**. Также можем создать свой каталог и создать своё ТП, указав путь к нашему каталогу. Попробуем на практике.

---

[43] DDL URL: https://en.wikipedia.org/wiki/Data_definition_language (дата обращения 10.03.2026) [6]

[44] create database URL: https://www.postgresql.org/docs/18/sql-createdatabase.html (дата обращения 10.03.2026) [7]

Создадим подкаталог в каталоге /home/ под суперпользователем Linux **root** (Linux-пользователю postgres прав не хватит), для этого выйдем из пользователя **postgres** и повысим свои права через **sudo**. Поменяем его владельца на пользователя Linux **postgres**:

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

Переключимся на пользователя Linux postgres и в новом месте создадим каталог для нашего ТП:

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

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

```
psql
CREATE TABLESPACE [45] ts location '/home/postgres/tmptblspc';
```

Посмотрим на список ТП:

```
\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 ТП по умолчанию:

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

Подключимся к созданной БД:

```
\c app
```

[45] create tablespace URL: https://www.postgresql.org/docs/current/sql-createtablespace.html (дата обращения 10.03.2026) [8]

Чтобы посмотреть дефолтный tablespace, выполним:

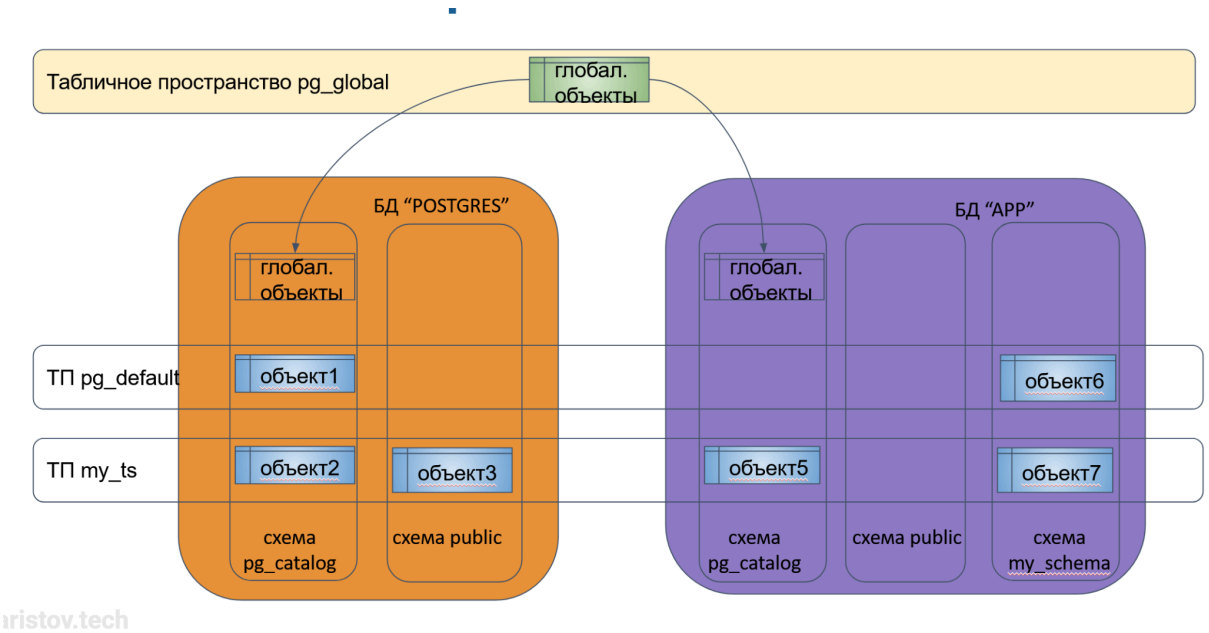
```
\l+
```

```
                                                                                                List of databases
   Name    |  Owner   | Encoding | Locale Provider | Collate |  Ctype  | Locale | ICU Rules |   Access privileges   |  Size   | Tablespace
on
-----------+----------+----------+-----------------+---------+---------+--------+-----------+-----------------------+---------+------------
------------------
 app       | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8 |        |           |                       | 7742 kB | ts
 book      | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8 |        |           |                       | 7585 kB | pg_default
 postgres  | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8 |        |           |                       | 7670 kB | pg_default
onnection database
 template0 | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8 |        |           | =c/postgres          +| 7670 kB | pg_default
ase
           |          |          |                 |         |         |        |           | postgres=CTc/postgres |         |
 template1 | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8 |        |           | =c/postgres          +| 7742 kB | pg_default
 databases
           |          |          |                 |         |         |        |           | postgres=CTc/postgres |         |
 thai      | postgres | UTF8     | libc            | C.UTF-8 | C.UTF-8 |        |           |                       | 589 MB  | pg_default
(6 rows)
```

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

Без конкретного указания ТП, при создании таблицы в БД app, она автоматически будет расположена в tablespace **ts**. Конечно, можно точно указать, в каком ТП конкретно хранить таблицу.

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

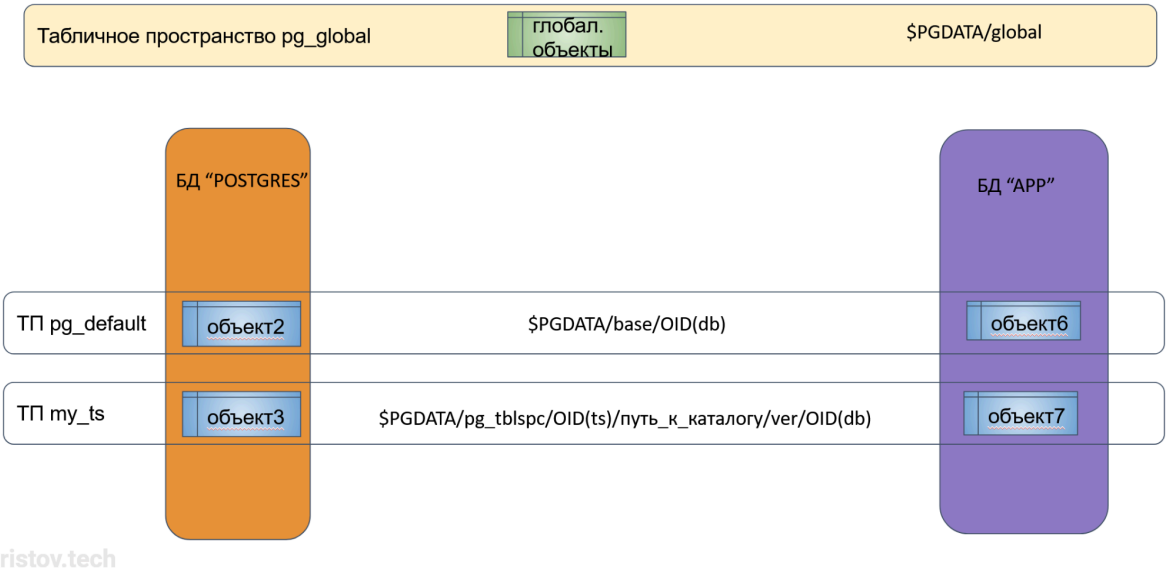


Видим, что каждый отдельный объект (таблица, индекс, материализованное представление) может принадлежать только к одному ТП (**pg_default** или другое), одной базе данных (**app** или **postgres**) и одной схеме (у каждой БД свой набор схем, про них будем говорить в теме про логическое устройство PostgreSQL). И только глобальные/системные объекты физически хранятся отдельно в ТП **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 ТП, но нужно понимать, что это будет физическое перемещение файлов между каталогами, требует эксклюзивной блокировки и на это время доступа к объекту не будет. Можно обойти ограничение, используя ряд утилит, которые рассмотрим в главе 22.

Ещё 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:



Важно понимать разницу между горячими и холодными данными. Уже на этом этапе рекомендовано самые востребованные данные положить на самые быстрые накопители, а очень редко используемые – на дешёвые и медленные.

Ещё варианты оптимизации производительности:

- распараллелить нагрузку на разные таблицы по разным рейд-массивам/NVMe/другим системам хранения, в том числе с использованием секционирования – перенос секций
- вынести WAL на отдельный диск (глава 8)
- ну и самый быстрый вариант – монтируем память в дисковую подсистему и подключаем самые нагруженные данные, например, индексы или материализованные представления
  - *важно! после перезагрузки ВМ эти данные необходимо будет создать заново*

- ➢ для чувствительных данных должна быть какая-то синхронная таблица на диске для постоянного хранения
- ❖ так как по умолчанию в классических файловых системах сжатие не поддерживается, можно использовать файловые системы ZFS[46] / Btrfs[47] для экономии места
- ❖ при большом количестве файлов необходимо также тюнить параметры файловой системы и PostgreSQL по максимально допустимому одновременному количеству открытых файлов
- ❖ есть ещё много вариантов улучшений, но о них нужно говорить подробнее

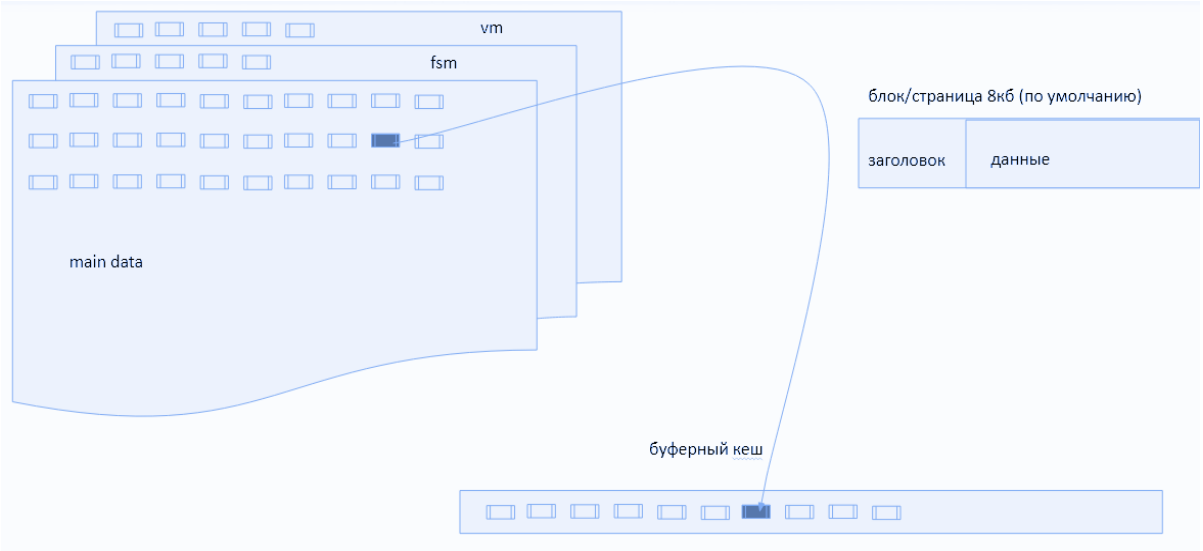
Теперь рассмотрим, как физически хранятся сами таблицы.

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:

[46] ZFS URL: https://en.wikipedia.org/wiki/ZFS (дата обращения 02.03.2026) [9]
[47] Btrfs URL: https://en.wikipedia.org/wiki/Btrfs (дата обращения 02.03.2026) [10]



Обратим внимание, что работа с данными идёт не побайтно, а блоками/страницами по 8 Кб[48] (размер задаётся при инициализации кластера, обычно никто не меняет, так как требует специальной сборки кластера из исходных кодов). Нужные данные извлекаются из файла и загружаются в буферный кэш постранично (более подробно как в PostgreSQL организована работа со слоями и кэшем, будем разбирать в следующих темах).

Необходимо заметить, что по умолчанию PostgreSQL старается всю строку нашей таблицы поместить в 8 Кб, и если встречаются длинные реквизиты (например, строки), то используется специальная TOAST-таблица (The Oversized-Attribute Storage Technique) для хранения таких данных. Тут есть свои тонкости:

- используется схема **pg_toast**
- поддержана 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
- подробно рассмотрим в теме 12

Собственно, в файловой системе созданные каталоги выглядят следующим образом:

```
cd tmptblspc/
ls
cd PG_18_202506291/
```

[48] КБ – килобайты

```
ls -la
```

```
postgres@Aeuge:/home/postgres$ cd tmptblspc/
postgres@Aeuge:/home/postgres/tmptblspc$ ls
PG_18_202506291
postgres@Aeuge:/home/postgres/tmptblspc$ cd PG_18_202506291/
postgres@Aeuge:/home/postgres/tmptblspc/PG_18_202506291$ ls -la
total 12
drwx------ 3 postgres postgres 4096 Feb 26 15:52 .
drwx------ 3 postgres postgres 4096 Feb 26 15:52 ..
drwx------ 2 postgres postgres 4096 Feb 26 15:52 16519
postgres@Aeuge:/home/postgres/tmptblspc/PG_18_202506291$
```

Видим каталог с БД с OID 16519, как и в списке БД:

```
psql -c "SELECT oid, datname, dattablespace FROM pg_database;"
```

```
postgres@Aeuge:/home/postgres/tmptblspc/PG_18_202506291$ psql -c "SELECT oid,
  oid  |  datname  | dattablespace
-------+-----------+---------------
     5 | postgres  |          1663
 16388 | thai      |          1663
     1 | template1 |          1663
     4 | template0 |          1663
 16518 | book      |          1663
 16519 | app       |         16517
(6 rows)
```

Зайдём в него и посмотрим, как выглядят таблицы в файловой системе:

```
cd 16519
ls -la
```

```
-rw------- 1 postgres postgres  16384 Feb 26 15:57 2753
-rw------- 1 postgres postgres  24576 Feb 26 15:57 2753_fsm
-rw------- 1 postgres postgres   8192 Feb 26 15:57 2753_vm
-rw------- 1 postgres postgres  16384 Feb 26 15:57 2754
-rw------- 1 postgres postgres  16384 Feb 26 15:57 2755
```

Видим, что размеры файлов кратны 8 КБ.

Давайте создадим свои таблицы в нашем новом табличном пространстве и 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** – 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');
```

```
app=# SELECT pg_relation_filepath('test2');
 pg_relation_filepath
----------------------
 base/16519/24579
(1 row)
```

Видим OID нашей таблицы и посмотрим на него сначала изнутри БД, сделав запрос к системной таблице **pg_class**:

```
SELECT OID, relname, relnamespace FROM pg_class WHERE OID=24579;
```

```
app=# SELECT OID, relname, relnamespace FROM pg_class WHERE OID=24579;
  oid  | relname | relnamespace
-------+---------+--------------
 24579 | test2   |         2200
(1 row)
```

И посмотрим теперь из файловой системы – из psql можно выполнять Linux-команды, используя префикс \! :

```
\! ls -la /var/lib/postgresql/18/main/pg_tblspc
```

```
app=# \! ls -la /var/lib/postgresql/18/main/pg_tblspc
total 8
drwx------  2 postgres postgres 4096 Feb 26 15:52 .
drwx------ 19 postgres postgres 4096 Feb 27 13:08 ..
lrwxrwxrwx  1 postgres postgres   24 Feb 26 15:52 16517 -> /home/postgres/tmptblspc
```

Видим жёсткую ссылку[49] на созданный выше каталог.

Выведем список всех файлов в БД и перешлём его на фильтр grep через прямую черту | :

```
\! ls -la
/var/lib/postgresql/18/main/pg_tblspc/16517/PG_18_202506291/16
519/ | grep 24579
```

И не увидим нашу таблицу! Всё потому, что переместили её в дефолтный ТП. Перенесём её в ТП **ts**:

```
ALTER TABLE test2 SET tablespace ts;
```

Повторим запрос и снова не увидели её!

Потому что при переносе изменяется OID!

```
SELECT pg_relation_filepath('test2');
```

```
app=# SELECT pg_relation_filepath('test2');
           pg_relation_filepath
------------------------------------------
 pg_tblspc/16517/PG_18_202506291/16519/24583
(1 row)
```

```
\! ls -la
/var/lib/postgresql/18/main/pg_tblspc/16517/PG_18_202506291/16
519/ | grep 24583
```

```
app=# \! ls -la /var/lib/postgresql/18/main/pg_tblspc/16517/PG_18_202506291/16519/ | grep 24583
-rw------- 1 postgres postgres      0 Feb 27 13:37 24583
```

Размер таблицы 0, потому что там нет данных. Добавим строку и посмотрим ещё раз:

[49] жёсткая ссылка URL: https://en.wikipedia.org/wiki/Hard_link (дата обращения 16.03.2026) [11]

```
INSERT INTO test2 VALUES ('1');
```

```
app=# INSERT INTO test2 VALUES ('1');
INSERT 0 1
app=# \! ls -la /var/lib/postgresql/18/main/pg_tblspc/16517/PG_18_202506291/16519/ | grep 24583
-rw------- 1 postgres postgres   8192 Feb 27 13:39 24583
```

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

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 – 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
- другие файловые системы и особенности работы разбираются в третьей главе моей книги по оптимизации PostgreSQL 16[50]

Воспроизведём на практике ситуацию, когда умирает физический носитель с табличным пространством.

Здесь есть два варианта:

- табличное пространство (ТП) было задано для базы данных по умолчанию
- отдельные объекты были размещены в таком ТП

Воспользуемся уже существующей базой данных **app**.

Создадим ещё одно табличное пространство под Linux-пользователем postgres, так как на родительский каталог права есть, проблем не должно быть:

```
exit
cd /home/postgres
mkdir tmptblspc2
psql -d app
CREATE tablespace ts2 location '/home/postgres/tmptblspc2';
```

[50] PostgreSQL 16: лучшие практики оптимизации. – М.: ООО «Сам Полиграфист», 2024. – 316 с

```
postgres@Aeuge:/home/postgres$ mkdir tmptblspc2
postgres@Aeuge:/home/postgres$ psql -d app
psql (18.2 (Ubuntu 18.2-1.pgdg24.04+1))
Type "help" for help.

app=# CREATE tablespace ts2 location '/home/postgres/tmptblspc2';
CREATE TABLESPACE
```

Переместим таблицу test во вновь созданное ТП:

```
ALTER TABLE test2 SET TABLESPACE ts2;
SELECT tablename, tablespace FROM pg_tables WHERE schemaname = 'public';
```

```
app=# ALTER TABLE test2 SET TABLESPACE ts2;
ALTER TABLE
app=# SELECT tablename, tablespace FROM pg_tables WHERE schemaname = 'public';
 tablename | tablespace
-----------+------------
 test      | pg_default
 test2     | ts2
(2 rows)
```

Выйдем из psql и удалим каталог с новым ТП:

```
exit
rm -rf tmptblspc2
```

Зайдём в psql и проверим доступ к БД и список таблиц:

```
psql -d app
\dt+
```

```
postgres=# \c app
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)
```

Вроде, таблица на месте…

Но при попытке получить к ней доступ получим следующую ошибку:

```
SELECT * FROM test2;
```

```
app=# SELECT * FROM test2;
ERROR:  could not open file "pg_tblspc/32840/PG_18_202506291/16519/32841": No such file or directory
```

Логично, ведь физически каталога теперь не существует.

Значительно более печальная ситуация произойдёт, если удалить дефолтное ТП для БД (аналогично может быть и ситуация с монтированием сетевого каталога).

Удалим ТП ts и попытаемся зайти в БД app:

```
exit
rm -rf tmptblspc
psql
```

И тут возможны две ситуации. Первая - сразу сломается PostgreSQL:

```
postgres@Aeuge:/home/postgres$ psql
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: No such file or directory
        Is the server running locally and accepting connections on that socket?
postgres@Aeuge:/home/postgres$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory              Log file
18  main    5432 down   postgres /var/lib/postgresql/18/main  /var/log/postgresql/postgresql-18-main.log
```

Запустить кластер тоже не получится:

```
postgres@Aeuge:/home/postgres$ pg_ctlcluster 18 main start
Warning: the cluster will not be running as a systemd service. Consider using systemctl:
  sudo systemctl start postgresql@18-main
Error: /usr/lib/postgresql/18/bin/pg_ctl /usr/lib/postgresql/18/bin/pg_ctl start -D /var/lib/postgresql/18/main -l /var/log/postgresql/postgresq
-c config_file="/etc/postgresql/18/main/postgresql.conf"  exited with status 1:
2026-03-03 14:26:47.200 +07 [2624] LOG:  starting PostgreSQL 18.2 (Ubuntu 18.2-1.pgdg24.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 13
13.3.0, 64-bit
2026-03-03 14:26:47.200 +07 [2624] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2026-03-03 14:26:47.203 +07 [2624] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2026-03-03 14:26:47.207 +07 [2624] LOG:  could not open directory "pg_tblspc/32840/PG_18_202506291": No such file or directory
2026-03-03 14:26:47.207 +07 [2624] LOG:  could not open directory "pg_tblspc/16517/PG_18_202506291": No such file or directory
2026-03-03 14:26:47.212 +07 [2630] LOG:  database system was interrupted while in recovery at 2026-03-03 14:20:45 +07
2026-03-03 14:26:47.212 +07 [2630] HINT:  This probably means that some data is corrupted and you will have to use the last backup for recovery.
2026-03-03 14:26:47.442 +07 [2630] LOG:  could not stat file "pg_tblspc/32840": No such file or directory
2026-03-03 14:26:47.442 +07 [2630] LOG:  could not stat file "pg_tblspc/16517": No such file or directory
2026-03-03 14:26:47.444 +07 [2630] LOG:  database system was not properly shut down; automatic recovery in progress
2026-03-03 14:26:47.444 +07 [2630] LOG:  could not open directory "pg_tblspc/32840/PG_18_202506291": No such file or directory
2026-03-03 14:26:47.444 +07 [2630] LOG:  could not open directory "pg_tblspc/16517/PG_18_202506291": No such file or directory
2026-03-03 14:26:47.447 +07 [2630] LOG:  could not open directory "pg_tblspc/32840/PG_18_202506291": No such file or directory
2026-03-03 14:26:47.447 +07 [2630] LOG:  could not open directory "pg_tblspc/16517/PG_18_202506291": No such file or directory
2026-03-03 14:26:47.447 +07 [2630] LOG:  redo starts at 0/D79E4908
2026-03-03 14:26:47.447 +07 [2630] FATAL:  could not create directory "pg_tblspc/16517": File exists
2026-03-03 14:26:47.447 +07 [2630] CONTEXT:  WAL redo at 0/D79E4A48 for XLOG/FPI_FOR_HINT: ; blkref #0: rel 16517/16519/1249, blk 16 FPW
2026-03-03 14:26:47.451 +07 [2624] LOG:  startup process (PID 2630) exited with exit code 1
2026-03-03 14:26:47.451 +07 [2624] LOG:  terminating any other active server processes
2026-03-03 14:26:47.452 +07 [2624] LOG:  shutting down due to startup process failure
2026-03-03 14:26:47.453 +07 [2624] LOG:  database system is shut down
pg_ctl: could not start server
```

Второй вариант – в старых версиях даст зайти, но обратиться к БД не даст.

```
\c app
```

```
postgres@postgres14:/home/postgres$ rm -rf tmptblspc
postgres@postgres14:/home/postgres$ psql
psql (14.0 (Ubuntu 14.0-1.pgdg21.04+1))
Type "help" for help.

postgres=# \c app
connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL:  database "app" does not exist
DETAIL:  The database subdirectory "pg_tblspc/16384/PG_14_202107181/16385" is missing.
Previous connection kept
```

Вообще не можем зайти в БД, так как оказались недоступны системные объекты, которые должны присутствовать в каждой БД.

Обратите внимание, что также потеряли доступ к таблице **test**, которая находится в схеме **pg_default**. Чтобы достать оттуда данные, придётся изрядно постараться.

В любом случае стандартным механизмом восстановления информации является или созданная заранее реплика, откуда эти данные сможем достать, или вовремя созданный бэкап. Более подробно это будет разбираться в соответствующих главах.

А с поломанной БД остаются два варианта: на старой версии PostgreSQL – удалить БД:

```
DROP DATABASE app;
```

Обратите внимание, что для удаления БД необходимо будет ещё перезапустить кластер (`pg_ctlcluster 18 main restart`*), так как остаются открытые файлы к несуществующему каталогу.*

На новой версии – восстановить из резервной копии (глава 19) или, как минимум, создать точку монтирования на месте удалённого каталога и вернуться к предыдущему варианту с удалением БД.

```
postgres@Aeuge:/home/postgres$ psql
psql (18.2 (Ubuntu 18.2-1.pgdg24.04+1))
Type "help" for help.

postgres=#      DROP DATABASE app;
DROP DATABASE
```

И не забыть удалить регистрацию ТП.

```
DROP TABLESPACE ts;
DROP TABLESPACE ts2;
```

Мы рассмотрели, как физически расположены и хранятся файлы БД. Настройки PostgreSQL будут рассмотрены в главе 11.

Посмотрим, как PostgreSQL выглядит в процессах и начнём с их порождения.

Первый и основной процесс PostgreSQL – **postgres server process:**

- называется postgres
- запускается при старте сервиса

- порождает все остальные процессы, используя клон процесса – fork[51]
- создаёт shared memory
- слушает TCP[52] и Unix socket[53]

background processes:

- запускаются основным процессом PostgreSQL при старте сервиса
- выделенная роль у каждого процесса (будем рассматривать тонкости 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** – демон (фоновый процесс), отвечающий за периодический запуск vacuum
  - **archiver** – архивация и репликация WAL
  - **statscollector** – сбор статистики использования по сессиям и таблицам
  - **ioworker0-2** – фоновые процессы для новой асинхронной системы ввода/вывода (подробнее в первой главе), появились только в PostgreSQL 18

Остальные порождаемые им процессы это **backend processes**. Каждый процесс:

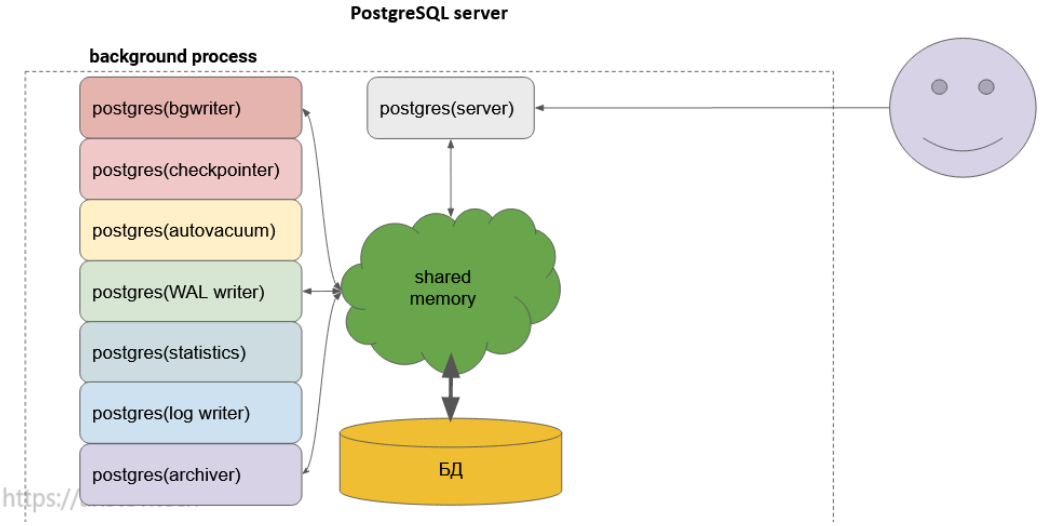
- сейчас тоже называется postgres
- запускается основным процессом PostgreSQL
- обслуживает клиентскую сессию
- работает, пока сессия активна
- максимальное количество определяется параметром max_connections (по умолчанию 100). После 600 начинается значительная деградация, подробно рассмотрим в главе 6

После того, как процессы клонированы, 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, должен же

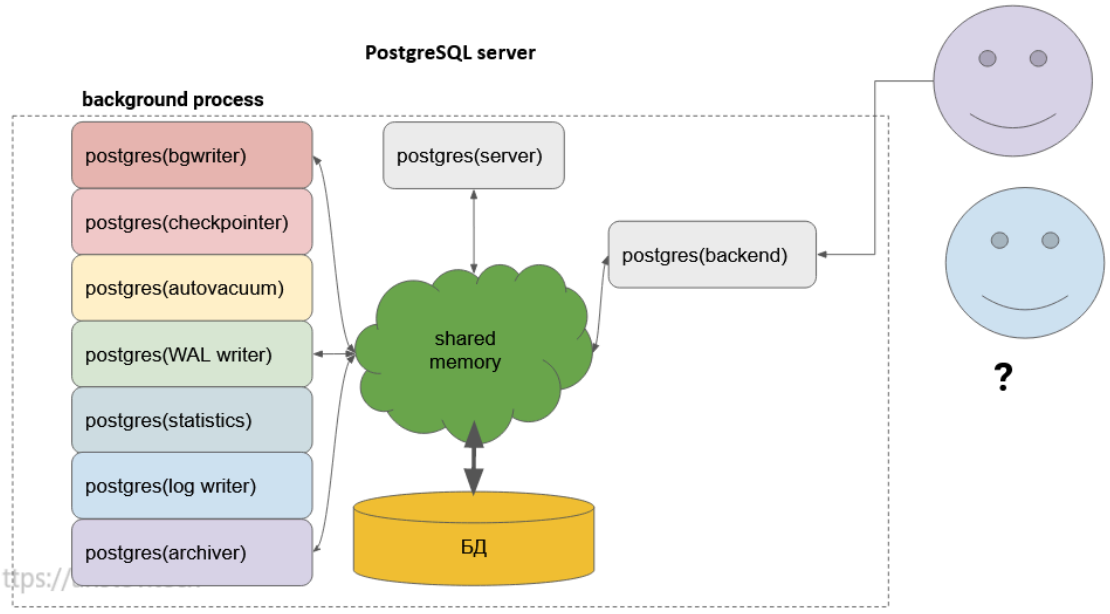
---

[51] fork URL: https://man7.org/linux/man-pages/man2/fork.2.html (дата обращения 20.03.2026) [12]
[52] TCP URL: https://en.wikipedia.org/wiki/Transmission_Control_Protocol (дата обращения 20.03.2026) [13]
[53] unix socket URL: https://en.wikipedia.org/wiki/Unix_domain_socket (дата обращения 20.03.2026) [14]

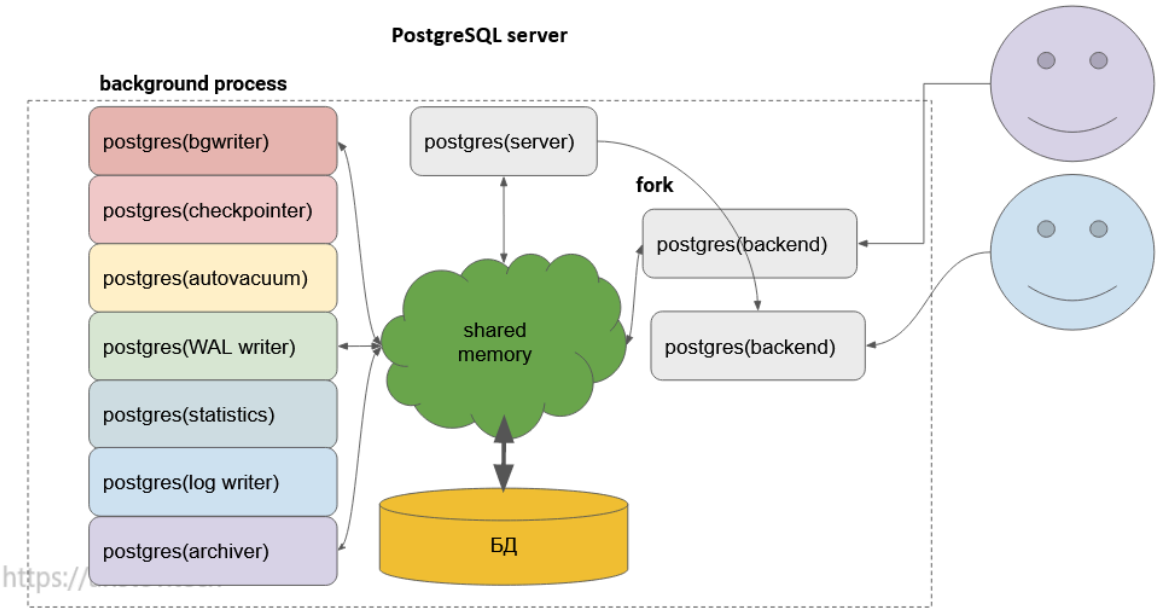
кто-то работать. Приходит к нам клиент (psql, java и так далее) и пытается подключиться к PostgreSQL:



Происходит **fork** процесса postgres, и дальше этот процесс **postgres backend** уже напрямую сам общается с **shared memory**. Далее приходит следующий клиент:



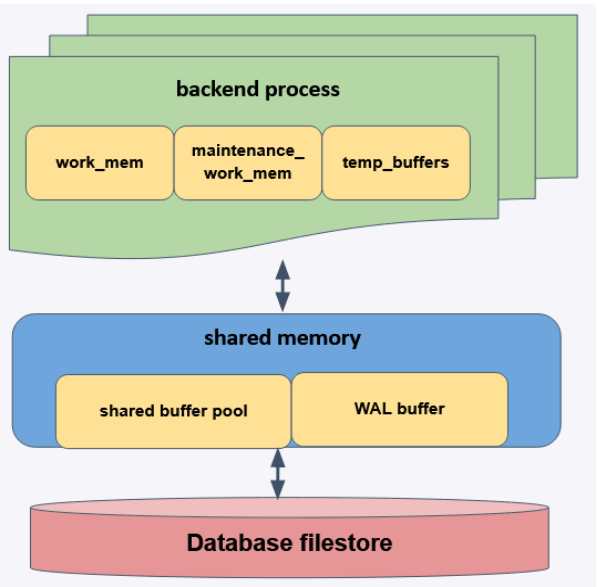
Снова клон процесса. На каждого пользователя свой бэкенд-процесс:



Клон процесса довольно быстр, но, всё же, это миллисекунды. Кроме этого, выделяется work_mem и temp_buffers на каждый процесс, что тоже несёт свои накладные расходы.

Получается, было бы оптимально не создавать клон процесса и выделять память под каждый запрос, а использовать специальную утилиту, которая как раз и будет держать постоянно висящие подключения и обслуживать посторонние сессии. И такая утилита называется pool connector (pooler или пулер).

Для каждого бэкенд-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*

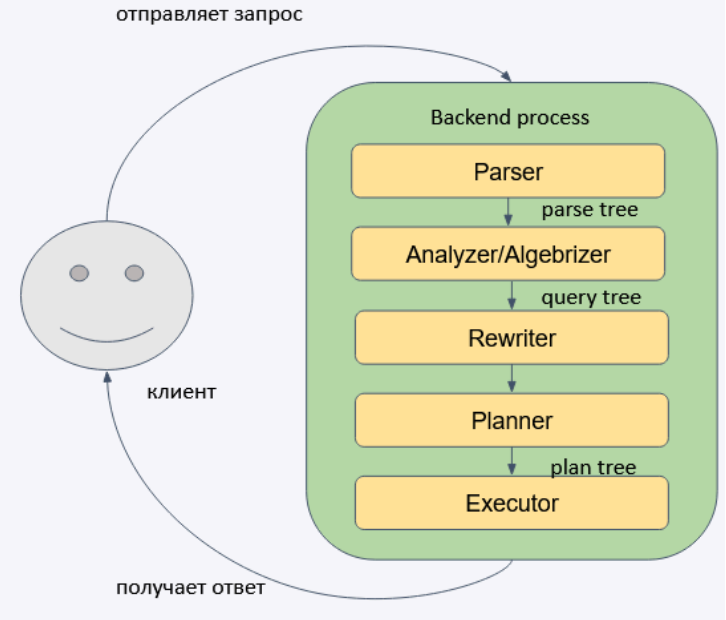
temp_buffers (8MB)

- используется на этапе выполнения для хранения временных таблиц

Все эти параметры настраиваются в зависимости от потребностей, подробнее в главе 11.

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 теперь понимаем, что процедура установки соединения в PostgreSQL очень дорогая. Именно поэтому рекомендуется использовать пулы подключений, например, **pgpool**[54]. Более подробно рассмотрим в главе 6.

Книга началась с возможностей версии 18, а как перейти на неё, если уже эксплуатируется другая версия PostgreSQL?

Обновление основной версии PostgreSQL.

Используется, когда решаем обновить устаревшую версию на более новую, например, повысить основную версию PostgreSQL с 17 до 18, классическим вариантом будет утилита **pg_upgradecluster**. Самым огромным недостатком данного метода является значительный даунтайм (время неработоспособности кластера), так как операция выполняется только на остановленном кластере. Время даунтайма зависит от объёма данных и выбранных опций обновления кластера (по умолчанию идёт копирование всех данных в новую версию кластера).

Для того, чтобы создать новый кластер PostgreSQL 17, необходимо установить эту его версию:

```
sudo DEBIAN_FRONTEND=noninteractive apt -y install postgresql-17
```

Посмотрим список кластеров:

```
pg_lsclusters
```

```
aeugene@Aeuge:/mnt/d/download$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory              Log file
18  main    5432 online postgres /var/lib/postgresql/18/main  /var/log/postgresql/postgresql-18-main.log
18  main2   5433 online postgres /var/lib/postgresql/18/main2 /var/log/postgresql/postgresql-18-main2.log
```

В новых версиях PostgreSQL кластер при установке автоматически НЕ создаётся, если обнаружены более новые версии.

Создадим новый кластер main версии 17.

```
pg_createcluster 17 main
```

```
aeugene@Aeuge:/mnt/d/download$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory              Log file
17  main    5434 down   postgres /var/lib/postgresql/17/main  /var/log/postgresql/postgresql-17-main.log
18  main    5432 online postgres /var/lib/postgresql/18/main  /var/log/postgresql/postgresql-18-main.log
18  main2   5433 online postgres /var/lib/postgresql/18/main2 /var/log/postgresql/postgresql-18-main2.log
```

[54] pgpool URL: https://www.pgpool.net/mediawiki/index.php/Main_Page (дата обращения 10.03.2026) [15]

Попробуем обновить кластер, не указывая версию (обновит на максимально доступную в ОС):

```
pg_upgradecluster 17 main
```

```
aeugene@Aeuge:/mnt/d/download$  pg_upgradecluster 17 main
Error: target cluster 18/main already exists
```

Ожидаемо, ничего не получилось, так как кластер по имени main уже существует для PostgreSQL 18 на данном инстансе.

Посмотрим помощь по команде и обдумаем варианты:

```
help pg_upgradecluster
```

```
PG_UPGRADECLUSTER(1)                    Debian PostgreSQL infrastructure                    PG_UPGRADECLUSTER(1

NAME
       pg_upgradecluster - upgrade an existing PostgreSQL cluster to a new major version.

SYNOPSIS
       pg_upgradecluster [-v newversion] oldversion name [newdatadir]

DESCRIPTION
       pg_upgradecluster upgrades an existing PostgreSQL server cluster (i. e. a collection of databases served by a postgres
       instance) to a new version specified by newversion (default: latest available version).  The configuration files of the
       old version are copied to the new cluster and adjusted for the new version.  The new cluster is set up to use data page
       checksums if the old cluster uses them.

       The cluster of the old version will be configured to use a previously unused port since the upgraded one will use the
       original port. The old cluster is not automatically removed. After upgrading, please verify that the new cluster indeed
       works as expected; if so, you should remove the old cluster with pg_dropcluster(8). Please note that the old cluster is
       set to "manual" startup mode, in order to avoid inadvertently changing it; this means that it will not be started
       automatically on system boot, and you have to use pg_ctlcluster(8) to start/stop it. See section "STARTUP CONTROL" in
       pg_createcluster(8) for details.
```

Выберем новый каталог для старой версии:

```
sudo pg_upgradecluster 17 main upgrade17
```

```
aeugene@Aeuge:/mnt/d/download$  pg_upgradecluster 17 main
Error: target cluster 18/main already exists
```

Ожидаемо, вариант не сработал, так как несмотря на то, что указали другой каталог – имя всё равно должно быть уникально.

Переименуем старую версию и посмотрим, что получилось:

```
sudo pg_renamecluster 17 main main17
```

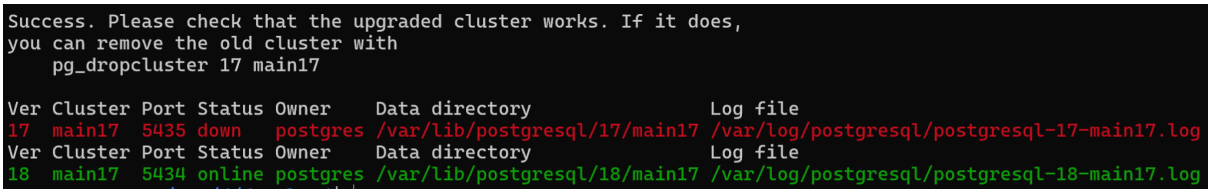
```
aeugene@Aeuge:/mnt/d/download$ pg_lsclusters
Ver Cluster Port Status Owner    Data directory              Log file
17  main17  5434 down   postgres /var/lib/postgresql/17/main17 /var/log/postgresql/postgresql-17-main17.log
18  main    5432 online postgres /var/lib/postgresql/18/main   /var/log/postgresql/postgresql-18-main.log
18  main2   5433 online postgres /var/lib/postgresql/18/main2  /var/log/postgresql/postgresql-18-main2.log
```

Всё готово для миграции.

Шифрование md5 и scram-sha256 несовместимы при апгрейде версии!

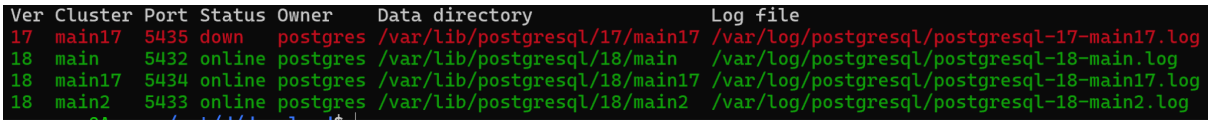
Обновим кластер до версии 18:

```
sudo pg_upgradecluster 17 main17
```



Посмотрим, что в итоге получилось:

```
pg_lsclusters
```



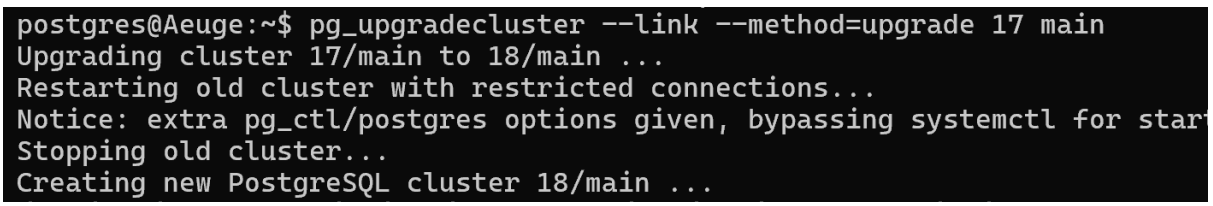
Всё! Теперь у нас новый кластер на версии 18, старый версии 17 можно удалять:

```
sudo pg_dropcluster 17 main17
```

Всё работает.

Минус этого варианта в полном копировании всего кластера – необходимо место x2 и большое время. Можно ускориться – использовать старые файлы с данными в виде жёстких ссылок, включив опции --link --method=upgrade:

```
pg_upgradecluster --link --method=upgrade 17 main
```



Важно заметить, что так обновить можно только PRIMARY-сервер, STANDBY нужно будет перезаливать через pg_basebackup.

Для удаления тестового кластера нужно сначала его остановить, а потом уже удалять:

```
sudo pg_ctlcluster 18 main17 stop
sudo pg_dropcluster 18 main17
```

Второй вариант попроще:

```
sudo pg_dropcluster 18 main17 --stop
```

В этой главе рассмотрели физический уровень – как и в каких файлах хранятся данные, как устроен PostgreSQL в физических процессах. Увидели, как можно оптимизировать работу, используя отдельные табличные пространства. Обновили наш кластер до новой версии. В следующей главе рассмотрим концепцию ACID и как организовывается параллельный доступ к данным в PostgreSQL.

** Задача на самостоятельное закрепление материала:*
Памяти у инстанса 4 GB (периодически приходил OOM killer[55])
max_connections = 1000 # (change requires restart)
shared_buffers = 6GB # min 128kB
work_mem = 16MB # min 64kB

Что не так с параметрами?

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/02/physical.sql

[55] Out of Memory Killer URL: https://habr.com/ru/company/southbridge/blog/464245/ (дата обращения 10.03.2026) [16]

3. ACID && MVCC

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 надёжность (durability)?

Ответ – транзакционные системы **OLTP**[56] – Online Transaction Processing. Они отвечают принципу **ACID**[57]**:**

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

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

- одна или более операций, выполняемые приложением
- которые переводят 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 есть следующие принципы (реализация долговечности **D**):

ARIES[58] – 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

---

[56] OLTP URL: https://en.wikipedia.org/wiki/Online_transaction_processing (дата обращения 13.03.2026) [1]

[57] ACID URL: https://en.wikipedia.org/wiki/ACID (дата обращения 13.03.2026) [2]

[58] ARIES URL: https://en.wikipedia.org/wiki/Algorithms_for_Recovery_and_Isolation_Exploiting_Semantics (дата обращения 13.03.2026) [3]

MVCC[59] – MultiVersion Concurrency Control. Конкурентный контроль мультиверсионности. Использует следующие механизмы:

- ❖ copy-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

Начнём с концепции **ACID**, а именно **транзакций**.

Рассмотрим классический сценарий.

Есть два счёта в банке – Светлана (500 Р) и Пётр (500 Р).

У нас нет транзакций и считаем все SQL-запросы атомарными. Светлана переводит 200Р Петру:

```
UPDATE account SET amount – 200 WHERE name = 'Svetlana';
-- ok
UPDATE account SET amount + 200 WHERE name = 'Petr';
-- ошибка
```

Первая команда завершилась удачно, а вторая не выполнилась, так как произошла какая-то ошибка – сетевая проблема, карта заблокирована у Петра или любая другая. Итог – деньги списали, но не зачислили. Что делать?

Правильно, использовать атомарную транзакцию:

```
BEGIN; -- начало транзакции аналог BEGIN TRANSACTION
UPDATE account SET amount – 200 WHERE name = 'Svetlana'; -- ok
UPDATE account SET amount + 200 WHERE name = 'Petr'; -- ошибка
COMMIT; -- команда подтверждения транзакции
```

Но транзакция не зафиксирует изменения, так как в одной из команд внутри транзакции была ошибка, а **транзакция** должна быть или **полностью** выполнена, или отменена. Соответственно, автоматически вместо COMMIT произойдёт ROLLBACK (отмена транзакции, как будто бы ничего и не было), так как была ошибка и деньги списаны не будут.

[59] MVCC URL: https://en.wikipedia.org/wiki/Multiversion_concurrency_control (дата обращения 13.03.2026) [4]

Опытный читатель спросит, а что произойдёт, если в этот же момент времени другая транзакция попытается получить доступ к этим же данным?

Мы должны обеспечить консистентность, изоляцию и атомарность. Для этого используется механизм MVCC.

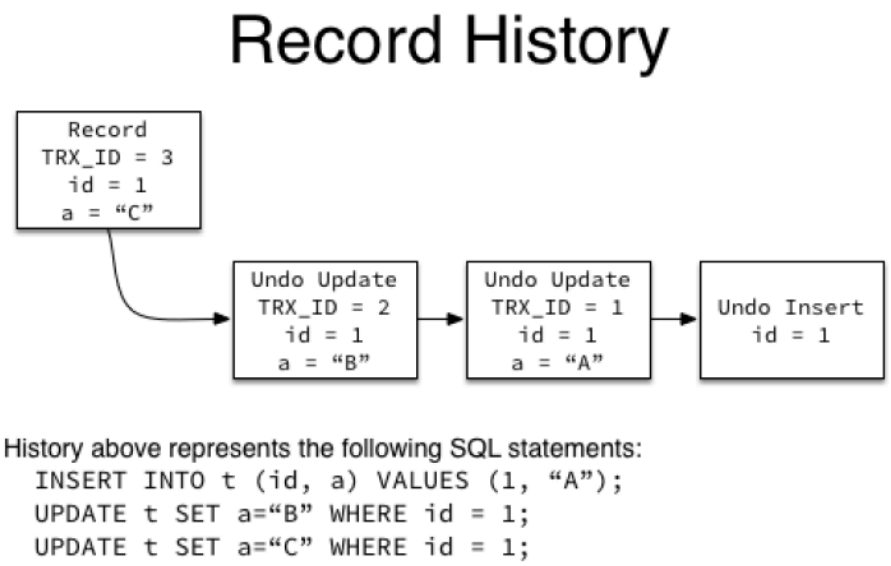
Механизм реализации MVCC в классических СУБД.

Стандартный механизм для Oracle Database, MySQL InnoDB и другие – использование **Undo** (rollback) **segment** (сегмент отката).

Особенности реализации:

- туда вносятся «противодействия». То есть, если в таблицу был внесён insert, то в сегмент отката вносится delete, delete – insert, update – вносится предыдущее значение строки. Блоки (и соответствующие данные undo) помечаются, как грязные, и переходят в redo log buffer (буфер воспроизведения изменений после сбоя). В журнал redo записываются не только инструкции, какие изменения стоит внести (redo), но и какие у них противодействия (undo)
- информация завершённых транзакций потихоньку оттуда вытесняется
  - *на процесс повлиять не можем*
- если rollback, нужно вернуть данные обратно в таблицу
  - *много лишней работы*
- если слишком много данных для вставки, удаления или обновления, сегмент может переполниться и произойдёт откат транзакции
  - *думаем, когда изменяем, добавляем и удаляем сотни миллионов строк, также ещё есть и параллельные транзакции*
- длинные транзакции могут стать проблемой

Графически можно представить так:



Но в PostgreSQL пошли своим уникальным путём.

Механизм реализации MVCC в PostgreSQL.

Tuple multi versioning – многоверсионность строк – **copy on write**.

Особенности работы:

- строки не удаляются в процессе обработки транзакций
- **создаются новые версии записей**
- дорого для операции update – происходит delete + insert
- нет кластерного индекса (только в виде одноразовой операции)
- необходим механизм уборки старых записей

Как это работает:

Создание версий данных: когда транзакция вносит изменения в базу данных, PostgreSQL не изменяет существующие строки, а создаёт новые версии строк с обновлёнными данными. Это позволяет другим транзакциям видеть старые версии данных, пока изменения не будут окончательно зафиксированы.

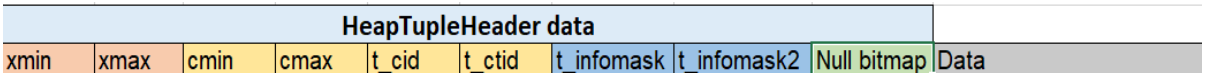
1. **Версионирование по времени**: каждая версия строки имеет информацию о времени начала и окончания её действия. Это позволяет транзакциям видеть данные в соответствии с моментом времени начала транзакции, что обеспечивает изоляцию.
2. **Уровни изоляции транзакций**: MVCC позволяет разным транзакциям работать на разных уровнях изоляции, таких как «читать

некоммитированные данные», «повторяемое чтение», «снимок» и «сериализуемость» (рассмотрим подробно в следующей главе). Это определяет, какие версии данных транзакция может видеть и какие блокировки должны быть установлены.

3. **Удаление устаревших данных**: PostgreSQL автоматически удаляет устаревшие версии данных, когда транзакция, которая их создала, успешно завершается (коммитится). Это помогает предотвратить накопление большого объёма старых данных.

MVCC является мощным механизмом, который позволяет базе данных обеспечивать высокую конкурентность и параллельность, минимизируя блокировки и обеспечивая высокий уровень изоляции для транзакций. Это делает его ключевой частью архитектуры PostgreSQL и других современных СУБД.

Рассмотрим как физически это реализовано. Данные хранятся поблочно и посмотрим, из чего состоит один блок:



В самом начале идёт служебный блок (23 байта), состоящий из:

- пары идентификаторов транзакций по 4 байта
- ряда информационных байтов
- пары информационных масок
- пустой битовой карты для будущего использования в следующих версиях PostgreSQL
- 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

Соответственно, когда выполняем операции вставки, обновления или удаления записи в PostgreSQL, 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()**

Следовательно, при добавлении записи происходят 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 нужно рассмотреть ещё дополнительные атрибуты строки – **infomask** содержит ряд битов, определяющих свойства данной версии:

- **xmin_commited**, **xmin_aborted** – xmin подтверждён или отменён
- **xmax_commited**, **xmax_aborted** – xmax подтверждён или отменён

Важно отметить поле **ctid** – ссылка 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 на диске, y – порядковый номер указателя в массиве.

Каждая транзакция может завершиться:

- **успешно** – после команды **commit**, для этого, в зависимости от типа операции, проставляются биты **xmin_commited**, **xmax_commited**
- **неуспешно** из-за, например, ошибки доступа к данным или вызове команды **rollback**

При неуспешном завершении нужно откатить все изменения, выполненные этой транзакцией. В случае с PostgreSQL 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 роллбека транзакции в PostgreSQL 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 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_tab
les WHERE relname = 'test';
 relname | n_live_tup | n_dead_tup | ratio%
---------+------------+------------+--------
 test    |          3 |          0 |      0
(1 row)
```

Обратите внимание на двойное двоеточие – специальный синтаксис в PostgreSQL для приведения типов данных.

Теперь, если обновить одну строчку из нашего набора, 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_tab
les WHERE relname = 'test';
 relname | n_live_tup | n_dead_tup | ratio%
---------+------------+------------+--------
 test    |          3 |          1 |     25
(1 row)
```

Видим, что 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 не видно запись со страницы под номером три.

Для более подробного изучения нужно установить расширение PostgreSQL **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)
```

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:

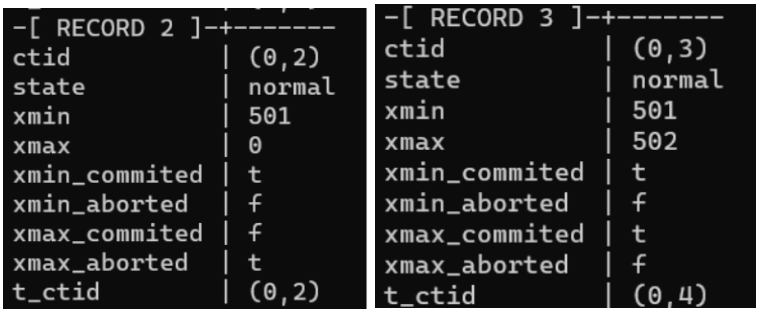
```
SELECT * FROM heap_page_items(get_raw_page('test',0)) \gx
```

```
-[ 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_commited,
       (t_infomask & 512) > 0  AS xmin_aborted,
       (t_infomask & 1024) > 0 AS xmax_commited,
       (t_infomask & 2048) > 0 AS xmax_aborted,
       t_ctid
FROM heap_page_items(get_raw_page('test',0)) \gx
```



Видим, что возле удалённой третьей записи стоит флаг **xmax_commited**, а у второй актуальной записи, наоборот, стоит флаг **xmax_aborted**. Соответственно, чтобы удалять мёртвые записи нужен какой-то механизм. И он называется вакуум (VACUUM). Подробнее этот механизм рассмотрим в главе 10.

Но что делать, если хотим максимальных гарантий согласованности? На самом деле можно регулировать уровни, на которых происходит изоляция транзакций друг от друга. Об этом в следующей главе.

Более глубокое погружение в материалы данной главы доступно в моём открытом вебинаре[60] «Особенности MVCC PostgreSQL». Запись занятия доступна на Rutube[61] и VKvideo[62].

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/03/mvcc.sql

[60] открытые лекции URL: https://aristov.tech/blog/otkrytye-lekczii-ot-aristov-tech/ (дата обращения 16.03.2026) [5]
[61] особенности MVCC URL: https://rutube.ru/video/b51b1efb04e84bd02722eb40bf311049/ (дата обращения 16.03.2026) [6]
[62] особенности MVCC URL: https://vkvideo.ru/video-212716752_456239042 (дата обращения 16.03.2026) [7]

4. Уровни изоляции транзакций

Уровни изоляции транзакций являются ключевым аспектом реляционных баз данных, включая PostgreSQL. Они определяют, как данные изолируются друг от друга при выполнении параллельных транзакций, обеспечивая согласованность и целостность данных.

PostgreSQL поддерживает стандартные уровни изоляции, определённые ANSI SQL, а также дополнительные режимы, обеспечивая гибкость и оптимизацию работы с данными. Уровни называются во взаимосвязи с аномалиями[63], которые допускаются или нет на том или другом уровне изоляции:

- **dirty read** («грязное» чтение). Транзакция 1 может читать строки изменённые, но ещё не зафиксированные транзакцией 2. ROLLBACK в транзакции 2 приведёт к тому, что транзакция 1 прочитает данные, которых никогда не существовало
- **non-repeatable read** (неповторяющееся чтение). После того, как транзакция 1 прочитала строку, транзакция 2 изменила или удалила эту строку и выполнила COMMIT. При повторном чтении этой же строки транзакция 1 видит, что строка изменена или удалена
- **phantom read** (фантомное чтение). Транзакция 1 прочитала набор строк по некоторому условию. Затем транзакция 2 добавила строки, также удовлетворяющие этому условию. Если транзакция 1 повторит запрос, она получит другую выборку строк
- **serialization anomaly** (аномалия сериализации). СУБД пытается выстроить транзакции последовательно во всех возможных комбинациях. При невозможности одного из вариантов происходит данная ошибка

Уровни изоляции и их особенности:

- **READ UNCOMMITTED (Чтение неподтверждённых данных):** Наименьший уровень изоляции, где транзакции видят неподтверждённые изменения других транзакций. Этот уровень обеспечивает максимальную параллельность, но может привести к непредсказуемым результатам из-за «грязного чтения»
- **READ COMMITTED (Чтение подтверждённых данных):** Большинство баз данных, включая **PostgreSQL, используют этот уровень по умолчанию**. Транзакции видят только подтверждённые изменения других

[63] изоляция транзакций URL: https://en.wikipedia.org/wiki/Isolation_(database_systems) (дата обращения 16.03.2026) [1]

транзакций. Это предотвращает «грязное чтение», но допускает «неповторяющееся чтение»

- **REPEATABLE READ (Повторяемое чтение):** В этом режиме транзакции видят только данные, которые были считаны на момент начала транзакции. Это предотвращает «грязное чтение» и «неповторяющееся чтение», но может привести к «фантомным» записям, когда другая транзакция вставляет новые записи
- **SERIALIZABLE (Сериализуемость):** Самый строгий уровень изоляции. Транзакции выполняются так, как если бы они выполнялись **последовательно**. Это предотвращает «грязное чтение», «неповторяющееся чтение» и «фантомные» записи. Однако это может привести к блокировкам и ухудшению производительности

Представим их соотношение в виде матрицы:

	dirty read	non-repeatable read	phantom read	serialization anomaly
Read Uncommitted	-	+	+	+
Read Committed	-	+	+	+
Repeatable Read	-	-	-	+
Serializable	-	-	-	-

На всех уровнях не допускается потеря зафиксированных изменений, то есть реализовывается буква D – Durability, но это если выставлен synchronous_commit в уровень local или выше (более подробно в главе 20).

Выбор уровня изоляции зависит от требований к согласованности данных и производительности. Более строгие уровни изоляции обеспечивают более высокую согласованность, но могут привести к блокировкам и ухудшению производительности. В то время как менее строгие уровни обеспечивают большую параллельность, но могут привести к непредсказуемым результатам.

Изменение уровня изоляции в PostgreSQL.

Установить уровень изоляции можно как кластер, на конкретную СУБД, так и на пользователя.

Также можно изменить уровень изоляции **для текущей транзакции** с помощью команды:

```
SET TRANSACTION ISOLATION LEVEL ...
```

Второй вариант — сразу начать транзакцию на нужном уровне изоляции, например:

```
BEGIN TRANSACTION ISOLATION LEVEL ...;
```

Уровни изоляции транзакций в PostgreSQL предоставляют средства для балансировки между согласованностью и производительностью при работе с данными. Выбор правильного уровня изоляции зависит от конкретных требований вашего приложения и ожидаемых характеристик работы с данными. Важно понимать особенности каждого уровня и тщательно анализировать, как он будет взаимодействовать с вашими данными и запросами.

Посмотрим на практике.

Подключимся одновременно с двух сеансов к одному инстансу PostgreSQL и в первой сессии выполним команды:

```
CREATE TABLE test (i serial, amount int);
INSERT INTO test(amount) VALUES (1);
INSERT INTO test(amount) VALUES (2);
show transaction isolation level;
```

```
postgres=# CREATE TABLE test (i serial, amount int);
INSERT INTO test(amount) VALUES (1);
INSERT INTO test(amount) VALUES (2);
show transaction isolation level;
CREATE TABLE
INSERT 0 1
INSERT 0 1
 transaction_isolation
-----------------------
 read committed
```

Начнём транзакцию на стандартном уровне изоляции PostgreSQL — **READ COMMITTED**:

```
BEGIN;
SELECT * FROM test;
```

Выполним команды из второй консоли, обновим одну строку:

```
BEGIN;
UPDATE test set amount = 3 WHERE i = 1;
COMMIT;
```

```
postgres=# BEGIN;
UPDATE test set amount = 3 WHERE i = 1;
COMMIT;
BEGIN
UPDATE 1
COMMIT
```

Проверим ещё раз состояние в первой консоли:

```
SELECT * FROM test;
COMMIT;
```

```
postgres=# BEGIN;
SELECT * FROM test;
BEGIN
 i | amount
---+--------
 1 |      1
 2 |      2
(2 rows)

postgres=*# SELECT * FROM test;
COMMIT;
 i | amount
---+--------
 2 |      2
 1 |      3
(2 rows)

COMMIT
```

Видим другое значение в той же транзакции — наблюдаем ситуацию «**неповторяющегося чтения**». Также все другие аномалии более высокого уровня будут встречены на этом уровне.

Теперь посмотрим, как себя поведёт PostgreSQL на уровне изоляции **REPEATABLE READ**.

В первой консоли выполним:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM test;
```

Во второй консоли:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
INSERT INTO test VALUES (8);
COMMIT;
```

И прочитаем ещё раз данные из первой консоли:

```
SELECT * FROM test;
COMMIT;
```

```
postgres=*# SELECT * FROM test;
COMMIT;
 i | amount
---+--------
 2 |      2
 1 |      3
(2 rows)

COMMIT
```

Аномалии «**фантомное чтение**» не наблюдаем.

Уровень **сериализации**[64] довольно сложен, в том числе для объяснения. Посмотрим классический пример. Допустим, есть таблица из четырёх значений:

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

Предположим, что сериализуемая транзакция A вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

И затем вставляет результат (30) в качестве значения в новую строку с классом = 2.

Одновременно сериализуемая транзакция B вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

А потом получает результат 300, который вставляет в новую строку с классом = 1.

Затем обе транзакции пытаются выполнить фиксацию.

Любая транзакция, которая первой зафиксирует результат – будет корректно обработана, вторая прервётся с ошибкой сериализации. Потому что если первая транзакция записала 30, то вторая должна записать 330 (если бы была выполнена последовательно), но у неё значение 300. И наоборот.

```
postgres=*# 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.
```

[64] уровень сериализации URL: https://www.postgresql.org/docs/current/transaction-iso.html#XACT-SERIALIZABLE (дата обращения 16.03.2026) [2]

Решением является переотправка отменённой транзакции.

Причём аналогичная ситуация на уровне **repeatable read** выполнилась бы без ошибок, но с аномалией сериализации.

На GitHub в файле isolation.sql приведены примеры на уровне serializable и repeatable read – оставляю на самостоятельное изучение.

В зависимости от установленного уровня изоляции, можно наблюдать те или иные аномалии. Повышение уровня изоляции несёт за собой дополнительные издержки, но позволяет исключить аномалии доступа к данным. Выше уровня **REPEATABLE READ** особо смысла увеличивать нет.

В этой главе рассмотрели уровни изоляции транзакций в PostgreSQL узнали о возможных аномалиях выполнения транзакций. В следующей главе рассмотрим логическую архитектуру PostgreSQL.

Подробнее в материалах открытого вебинара "Уровни изоляции транзакций в PostgreSQL". Запись занятия доступна на Rutube[65] и VKvideo[66].

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/04/isolation.sql

[65] уровни изоляции транзакций URL: https://rutube.ru/video/fa722ecbd84731e4c89613cad6759b22/ (дата обращения 16.03.2026) [3]
[66] уровни изоляции транзакций URL: https://vkvideo.ru/video-212716752_456239077 (дата обращения 16.03.2026) [4]

5. Логическая архитектура

Начнём с того, что посмотрим, какие сущности существуют как на физическом, так и на логическом уровне в PostgreSQL. Это база данных (**Database**) на логическом и каталог на физическом, а также отношение (**relation**) на логическом уровне и файлы на физическом.

Контейнер верхнего уровня в логическом устройстве – база данных (**Database**).

По умолчанию в любом кластере есть как минимум три БД:

- **postgres**
- **template0**
- **template1**

Присутствует на логическом и физическом уровне. На логическом уровне в одном экземпляре, на физическом в каждом **tablespace**, задействованном в БД – по умолчанию **pg_default** + те **tablespace**, где хранятся объекты нашей БД. Подробнее рассматривали в главе про физическое устройство.

Синтаксис команды создания базы данных[67] относительно прост:

CREATE DATABASE name
[[WITH] [OWNER [=] user_name]
[TEMPLATE [=] template]
[ENCODING [=] encoding]
[LOCALE [=] locale]
[LC_COLLATE [=] lc_collate]
[LC_CTYPE [=] lc_ctype]
[TABLESPACE [=] tablespace_name]
[ALLOW_CONNECTIONS [=] allowconn]
[CONNECTION LIMIT [=] connlimit]
[IS_TEMPLATE [=] istemplate]]

Соответственно, можем указать следующие параметры:

- **NAME** – имя создаваемой базы данных
- **OWNER** – имя роли, владельца новой базы данных. Если не указан – владельцем станет пользователь, выполняющий команду
- **TEMPLATE** – имя шаблона, из которого будет создаваться новая база данных. Можем указать любую другую существующую БД – произойдёт её копия – не рекомендованный способ на продуктовой БД, так как

[67] create database URL: https://www.postgresql.org/docs/18/sql-createdatabase.html (дата обращения 17.03.2026) [1]

требует эксклюзивного доступа к существующей БД и в это время невозможны коннекты извне. Default – **template1**

- **ENCODING** – кодировка символов в новой базе данных
  - *!!! не рекомендую использовать отличную от UTF-8 кодировку!!!*
  - *таким образом, если во всех ваших проектах используется Unicode, обеспечивается отсутствие конфликтов при работе с вашим ПО и БД по всему миру*
- **LOCALE**[68] – краткая запись для одновременной установки **LC_COLLATE** и **LC_CTYPE** – рекомендация оставить по умолчанию
  - *при различных кодировках в таблицах/индексах возможны большие проблемы с производительностью*
- **TABLESPACE** – имя табличного пространства, связываемого с новой базой данных. Это табличное пространство будет использоваться по умолчанию для объектов, создаваемых в этой базе. Default – **pg_default**
  - *подробнее разбирали во второй главе*
- **ALLOW_CONNECTIONS** – разрешить ли подключения к этой БД. Default – true – подключения принимаются. Если **false** – никто не сможет подключаться к этой базе данных. Возможно ограничить другими механизмами, например, **GRANT/REVOKE CONNECT** для конкретных пользователей/ролей
  - *обычно применяется для шаблонных баз данных*
- **CONNECTION LIMIT** – максимальное количество одновременных подключений к этой базе данных. Default – "**-1**" – без ограничений
- **IS_TEMPLATE** – если true, базу данных сможет клонировать любой пользователь с правами CREATEDB; в противном случае (по умолчанию) клонировать эту базу смогут только суперпользователи и её владелец. Default – **false**

Посмотрим на практике.

Создадим базу данных logical:

```
CREATE DATABASE logical;
```

```
postgres=# CREATE DATABASE logical;
CREATE DATABASE
postgres=#
postgres=# \c logical
You are now connected to database "logical" as user "postgres".
```

Посмотрим существующие базы данных на практике и их размеры:

[68] lc_locale & Lc_ctype URL: https://www.postgresql.org/docs/18/locale.html (дата обращения 17.03.2026) [2]

```
SELECT oid, datname, datistemplate, datallowconn FROM pg_database;
SELECT pg_size_pretty(pg_database_size('logical'));
```

```
logical=# SELECT oid, datname, datistemplate, datallowconn FROM pg_database;
  oid  |  datname  | datistemplate | datallowconn
-------+-----------+---------------+--------------
     5 | postgres  | f             | t
 16388 | thai      | f             | t
     1 | template1 | t             | t
     4 | template0 | t             | f
 16518 | book      | f             | t
 49152 | logical   | f             | t
(6 rows)

logical=# SELECT pg_size_pretty(pg_database_size('logical'));
 pg_size_pretty
----------------
 7742 kB
```

Видим, что почти пустая база данных занимает 8 МБ за счёт системных объектов.

Контейнер второго уровня – схемы (**Schema**).

В одной базе данных может быть множество схем, независимых от схем в других базах данных. Схема по умолчанию **public**.

CREATE SCHEMA[69] создаёт новую схему в текущей базе данных. Имя схемы должно отличаться от имён других существующих схем в текущей базе данных.

Схема представляет собой пространство имён (**namespace**) – она содержит именованные объекты (таблицы, индексы, функции и т.д.), **имена** которых **могут совпадать** с именами аналогичных объектов, существующих в других схемах этой базы данных.

Для обращения к объекту нужно либо «дополнить» его имя именем схемы в виде префикса, либо установить путь поиска (**search_path**), включающий требуемую схему. Команда **CREATE TABLE/VIEW**, в которой указывается неполное имя объекта, создаёт объект в текущей схеме (схеме, стоящей первой в переменной пути поиска. Узнать её позволяет функция `SHOW search_path;`).

[69] create schema URL: https://www.postgresql.org/docs/18/sql-createschema.html (дата обращения 17.03.2026) [3]

Чтобы создать схему, пользователь должен иметь право CREATE в текущей базе данных. Разумеется, на суперпользователей это условие не распространяется.

Рассмотрим синтаксис команд создания схемы:

CREATE SCHEMA schema_name [AUTHORIZATION role_specification] [schema_element [...]]

CREATE SCHEMA **IF NOT EXISTS** schema_name [AUTHORIZATION role_specification]

Рассмотрим основные параметры:

schema_name – имя создаваемой схемы. Если оно опущено, именем схемы будет **user_name**. Это имя **не может начинаться с pg_**, так как такие имена зарезервированы для системных схем.

user_name – имя пользователя (роли), назначаемого владельцем новой схемы. По умолчанию владельцем будет пользователь, выполняющий команды. Чтобы назначить владельцем создаваемой схемы другую роль, необходимо быть прямым или унаследованным членом этой роли, либо суперпользователем.

IF NOT EXISTS – если схема с таким именем уже существует, то вместо ошибки выдать предупреждение.

Правилом хорошего тона является использование данного указания для создания объектов при написании DDL-скриптов, чтобы не было неожиданных ошибок при их выполнении.

Посмотрим на практике.

Список схем в конкретной БД и текущая схема:

```
\dn
SELECT current_schema();
```

```
logical=# \dn
  List of schemas
  Name  |  Owner
--------+----------
 public | postgres
(1 row)

logical=# SELECT current_schema();
 current_schema
----------------
 public
(1 row)
```

Обратите внимание, что видим одну схему, а на самом деле есть ещё ряд системных, при этом они присутствуют по умолчанию и в строке поиска (**search path**[70]).

```
SELECT * FROM pg_namespace;
```

```
logical=# SELECT * FROM pg_namespace;
  oid  |      nspname       | nspowner |                          nspacl
-------+--------------------+----------+-------------------------------------------------------------
    99 | pg_toast           |       10 |
    11 | pg_catalog         |       10 | {postgres=UC/postgres,=U/postgres}
  2200 | public             |     6171 | {pg_database_owner=UC/pg_database_owner,=U/pg_database_owner}
 13273 | information_schema |       10 | {postgres=UC/postgres,=U/postgres}
```

Эти строки показывают схемы базы данных в PostgreSQL и права доступа к ним. Давайте разберём, зачем нужна каждая из них:

1. **public** (схема по умолчанию) – обычно здесь создаются пользовательские таблицы

- это "рабочее пространство" для ваших данных
- по умолчанию все новые таблицы создаются здесь
- удобна для небольших проектов или начала работы

Рекомендую НЕ использовать и лучше удалить.
Создавайте именованные схемы для своих проектов!

2. **pg_catalog** – системный каталог PostgreSQL

- хранит метаданные: информацию о всех таблицах, индексах, типах данных
- содержит системные таблицы (pg_class, pg_attribute и др.)
- обеспечивает работу SQL-стандарта INFORMATION_SCHEMA

3. **information_schema** – стандартизированный системный каталог

- предоставляет доступ к метаданным в стандартном SQL-формате
- удобна для кросс-платформенных приложений

[70] search path URL: https://www.postgresql.org/docs/18/ddl-schemas.html#DDL-SCHEMAS-PATH (дата обращения 17.03.2026) [4]

- содержит представления (views) с информацией о БД

4. **pg_toast** – служебная схема для больших объектов
- хранит "переполненные" данные (TOAST – подробнее в 12 главе)
- автоматически используется, когда данные в полях превышают размер страницы (8KB)
- позволяет хранить большие тексты, JSON, массивы эффективно

Расшифровка прав доступа:
- UC = USAGE + CREATE (право использовать и создавать объекты)
- =U = публичный доступ на USAGE
- postgres= = права для пользователя postgres
- pg_database_owner= = права для владельца БД

Что же такое **строка поиска (search_path)** и зачем она нужна?

Везде писать полные имена утомительно и часто всё равно лучше не привязывать приложения к конкретной схеме. Поэтому к таблицам обычно обращаются по **неполному имени**, состоящему просто из имени таблицы. Система определяет, какая именно таблица будет использоваться, используя **путь поиска**, который представляет собой список просматриваемых схем. Первая подходящая таблица, найденная в схемах пути, и будет использована. Если подходящая таблица не найдена, возникает ошибка, даже если таблица с таким именем есть в других схемах базы данных, не включённых в путь поиска.

Первая схема в пути поиска называется текущей. Эта схема будет использоваться не только при поиске, но и при создании объектов – она будет включать таблицы, созданные командой CREATE TABLE без указания схемы.

Посмотрим на строку поиска:

```
SHOW search_path;
```

```
postgres=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

Изменить этот параметр можем как в рамках сессии:

```
SET search_path TO [список_схем]
```

Так и на уровне отдельной базы данных:

```
ALTER DATABASE app SET search_path = public, another_schema;
```

А также в рамках кластера в файле конфигурации **postgresql.conf**.

Обратите внимание, что при поиске, до того как будут использоваться схемы, указанные в `search_path`*, будут обработаны ещё и схемы* `PG_CATALOG`, `PG_TOAST`, “`$user`”*.*

Есть ещё псевдосхема для временных таблиц **PG_TEMP** – тоже неявно участвует в поиске.

При создании отношений также будет присутствовать переменная с именем текущего пользователя **“$user”**.

Давайте это проверим.

Создадим новую таблицу **testL:**

```
CREATE TABLE testL(i int);
```

Посмотрим все таблицы в нашей БД:

```
\dt
```

```
logical=#       \dt
          List of tables
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | testl | table | postgres
(1 row)
```

Выберем данные из таблицы **pg_database**:

```
\d pg_database
```

```
logical=# \d pg_database
              Table "pg_catalog.pg_database"
     Column     |   Type    | Collation | Nullable | Default
----------------+-----------+-----------+----------+---------
 oid            | oid       |           | not null |
 datname        | name      |           | not null |
 datdba         | oid       |           | not null |
 encoding       | integer   |           | not null |
 datlocprovider | "char"    |           | not null |
 datistemplate  | boolean   |           | not null |
 datallowconn   | boolean   |           | not null |
 dathasloginevt | boolean   |           | not null |
 datconnlimit   | integer   |           | not null |
 datfrozenxid   | xid       |           | not null |
 datminmxid     | xid       |           | not null |
 dattablespace  | oid       |           | not null |
 datcollate     | text      | C         | not null |
 datctype       | text      | C         | not null |
 datlocale      | text      | C         |          |
 daticurules    | text      | C         |          |
 datcollversion | text      | C         |          |
 datacl         | aclitem[] |           |          |
Indexes:
    "pg_database_oid_index" PRIMARY KEY, btree (oid), tablespace "pg_global"
    "pg_database_datname_index" UNIQUE CONSTRAINT, btree (datname), tablespace "pg_global"
Tablespace: "pg_global"
```

И видим, что подставились таблицы из схемы **pg_catalog**. При этом видим, что используется tablespace **pg_global**.

Теперь создадим просто аналогичную таблицу:

```
CREATE TABLE pg_database (i int);
```

Посмотрим описание:

```
\d pg_database
```

```
logical=# CREATE TABLE pg_database (i int);
CREATE TABLE
logical=# \d pg_database
              Table "pg_catalog.pg_database"
     Column     |   Type    | Collation | Nullable | Default
----------------+-----------+-----------+----------+---------
 oid            | oid       |           | not null |
```

И увидим, что **опять будет таблица** из **pg_catalog**!

Чтобы вывести данные нашей новой таблицы, нужно конкретно указать схему:

```
\d public.pg_database
```

```
logical=# \d public.pg_database
             Table "public.pg_database"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 i      | integer |           |          |
```

Очередной случай. Создадим две таблицы, и вопрос – они создадутся в одной схеме или разных?

```
CREATE TABLE t1(i int);
CREATE SCHEMA postgres;
CREATE TABLE t2(i int);
```

Как думаете?

Правильно, в разных:

```
\dt
```

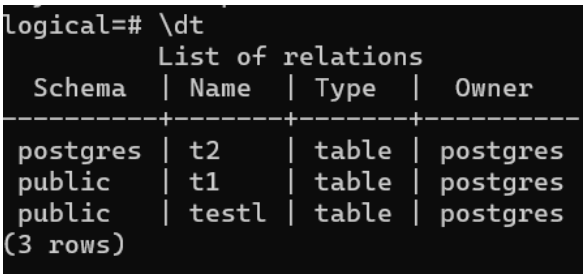
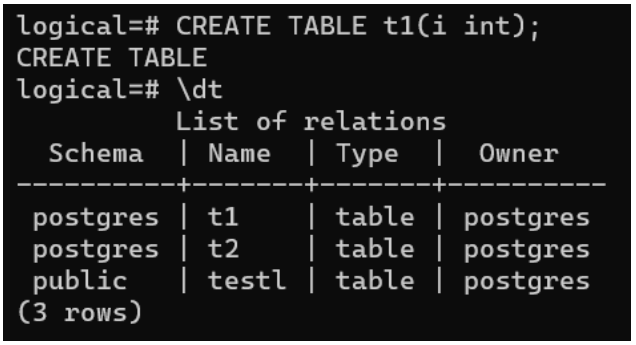


Таблица t2 оказалась в схеме **postgres**, так как **неявно присутствует '$user'** в строке поиска.

И теперь можем ещё раз создать таблицу t1, но теперь она будет уже в схеме postgres:

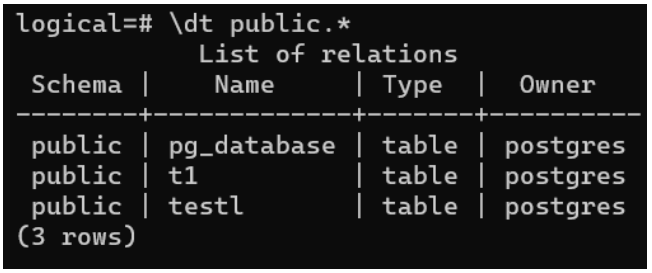
```
CREATE TABLE t1(i int);
\dt
```



Интересный побочный эффект – из списка исчезла таблица **public.t1**. Но физически она есть.

Чтобы на неё посмотреть, можем указать конкретную схему для вывода списка таблиц:

```
\dt public.*
```



Заодно увидели созданную нами ранее таблицу **pg_database**.

Чтобы избежать таких проблем – **желательно всегда полностью указывать схемы при указании таблицы**.

Также можем указывать неявные схемы в конце списка.

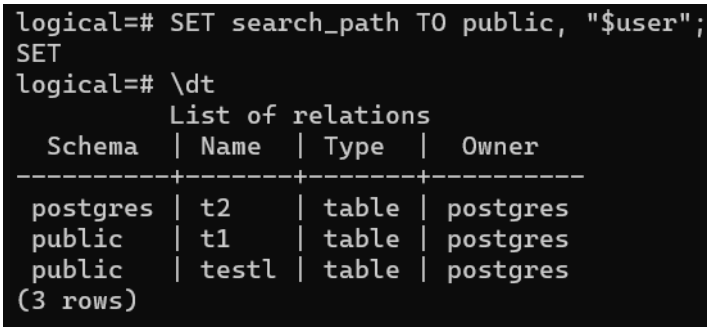
Например:

```
SET search_path TO public, "$user", pg_temp;
```

Также необходимо отзывать у пользователей права на дефолтные схемы типа **public**. Более подробно про права пользователя – в главе 7.

Давайте поменяем **search_path** на `public, "$user";` и заново сделаем запрос, какие есть таблицы в БД:

```
SET search_path TO public, "$user";
```



Видим, что теперь отображается таблица t1 из схемы public.

А если сделать так:

```
SET search_path TO public, "$user", pg_catalog;
\dt
```

```
logical=# SET search_path TO public, "$user", pg_catalog;
SET
logical=# \dt
            List of relations
  Schema  |    Name     | Type  |  Owner
----------+-------------+-------+----------
 postgres | t2          | table | postgres
 public   | pg_database | table | postgres
 public   | t1          | table | postgres
 public   | testl       | table | postgres
(4 rows)
```

То появилась и наша таблица **pg_database**.

А теперь магия – снова можем создать таблицу t1:

```
create temp table t1(i int);
\dt
```

```
logical=# create temp table t1(i int);
CREATE TABLE
logical=# \dt
              List of relations
  Schema   |    Name     | Type  |  Owner
-----------+-------------+-------+----------
 pg_temp_4 | t1          | table | postgres
 postgres  | t2          | table | postgres
 public    | pg_database | table | postgres
 public    | testl       | table | postgres
(4 rows)
```

Видим, что теперь таблица t1 по умолчанию находится в схеме **pg_temp**. Таким образом злоумышленники могут подменять временными таблицами настоящие.

Например, при неаккуратно настроенном доступе можно создать временную таблицу users, записать туда своего пользователя и получить полный доступ к БД для вашего приложения. Неаккуратно настроенный **search_path** создаст такую лазейку для злоумышленников!

Исправим ситуацию:

```
SET search_path TO public, "$user", pg_catalog, pg_temp;
```

```
logical=# SET search_path TO public, "$user", pg_catalog, pg_temp;
SET
logical=# \dt
            List of relations
  Schema  |    Name     | Type  |  Owner
----------+-------------+-------+----------
 postgres | t2          | table | postgres
 public   | pg_database | table | postgres
 public   | t1          | table | postgres
 public   | testl       | table | postgres
(4 rows)
```

Повторяю рекомендацию – используйте полные квалифицированные имена с указанием схемы. Или аккуратнее настраивайте search_path.

Третий уровень – уже непосредственно отношения (**relation**). Одной схеме может принадлежать от нуля и более отношений. Соответственно, каждое отношение принадлежит своей схеме и своей базе данных.

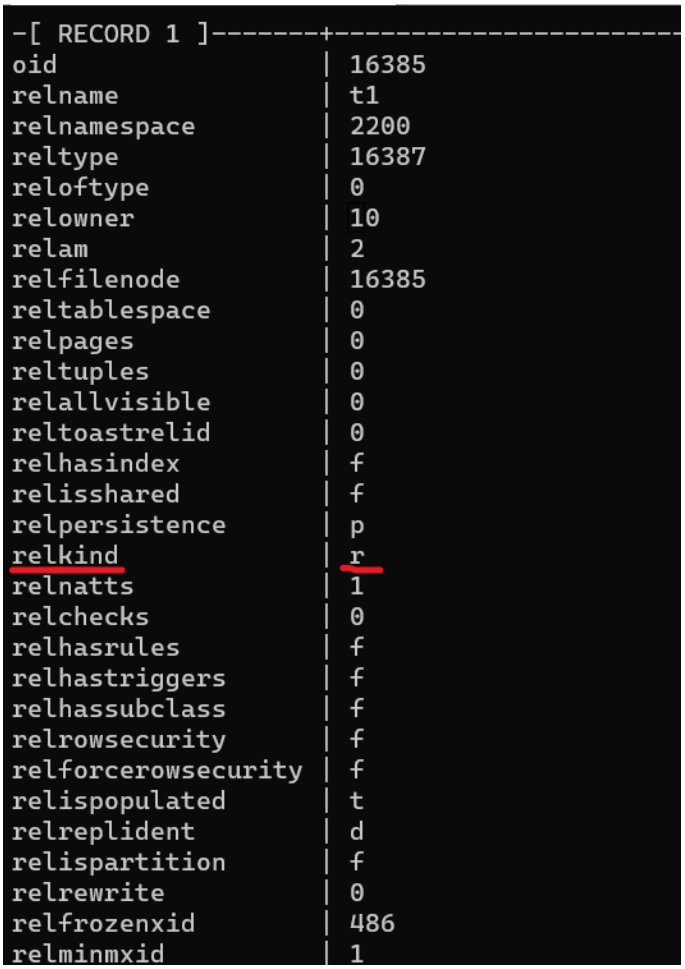
Давайте посмотрим, какие есть виды отношений (**relkind**):

- **r = ordinary table** – обычная таблица
- **i = index** – индекс
- **S = sequence** – последовательность
- **v = view** – представление
- **m = materialized view** – материализованное представление
- **c = composite type** – композитный тип[71]
- **t = TOAST table** – для хранения слишком больших строк
- **f = foreign table** – внешние таблицы[72]

Буква – тип, указанный в системной таблице **pg_class**, хранящей все типы объектов:

```
SELECT * FROM pg_class \gx
```

[71] композитный тип URL: https://www.postgresql.org/docs/18/rowtypes.html (дата обращения 17.03.2026) [5]
[72] внешние таблицы URL: https://www.postgresql.org/docs/18/sql-createforeigntable.html (дата обращения 17.03.2026) [6]

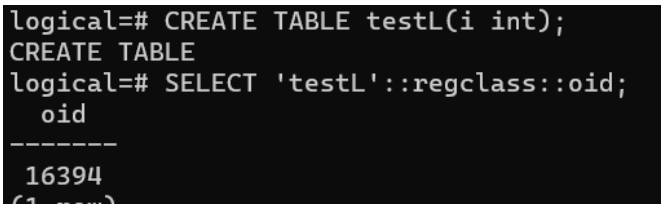


Видим:

- **relname** – имя объекта
- **relkind** – его тип – **r** (**таблица**)
- **OID (Object identifier) отношения** – идентификатор, по которому можем найти его в нашей файловой системе
- **OID схемы (namespace)** – аналогично для табличного пространства

Вспоминаем, что была таблица **testL.** Узнаем её **oid**:

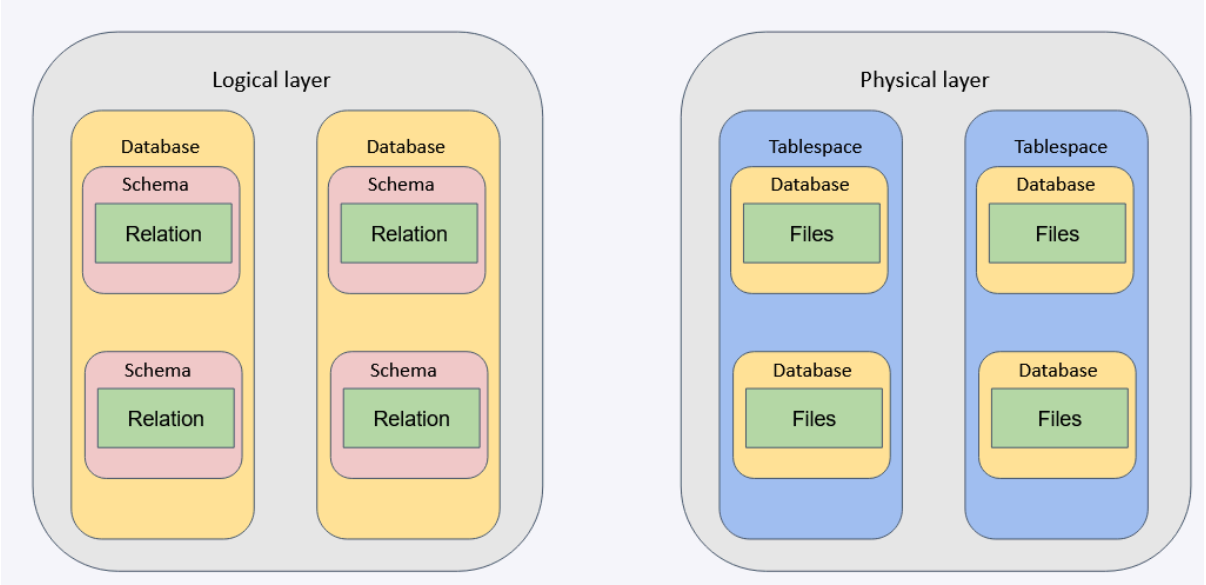
```
SELECT 'testL'::regclass::oid;
```



В последней строке преобразовали наш объект **таблица** к объекту **regclass** и достали оттуда тип данных **oid**.

Как выглядит объект на физическом уровне разбирали во второй главе.

Итак, рассмотрели три уровня логической модели PostgreSQL. Схематически логический и физический уровни можно изобразить так:



Видим ответ на первый вопрос нашей главы – как соответствуют физический и логический уровни:

- database <-> database
- relation <-> files

Есть возможность переносить таблицу между схемами – при этом меняется только запись в pg_class, физически данные на месте:

ALTER TABLE t2 SET SCHEMA public;

В этой главе рассмотрели логическое устройство PostgreSQL, а также базы данных, схемы и отношения. Увидели, как работает путь поиска и какие есть тонкие места при его настройке. В следующей главе рассмотрим работу с правами пользователя.

В качестве бонуса в конце главы посмотрим, какие на самом деле происходят запросы внутри встроенных команд psql:

```
\set ECHO_HIDDEN on
\l
\set ECHO_HIDDEN off
```

```
logical=# \set ECHO_HIDDEN on
logical=# \l
********* QUERY **********
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges"
FROM pg_catalog.pg_database d
ORDER BY 1;
**************************

                                 List of databases
   Name    |  Owner   | Encoding | Collate |  Ctype  |   Access privileges
-----------+----------+----------+---------+---------+-----------------------
 app       | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 book      | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 logical   | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 postgres  | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 template0 | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
           |          |          |         |         | postgres=CTc/postgres
 template1 | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
           |          |          |         |         | postgres=CTc/postgres
(6 rows)

logical=# \set ECHO_HIDDEN off
```

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/05/logical.sql

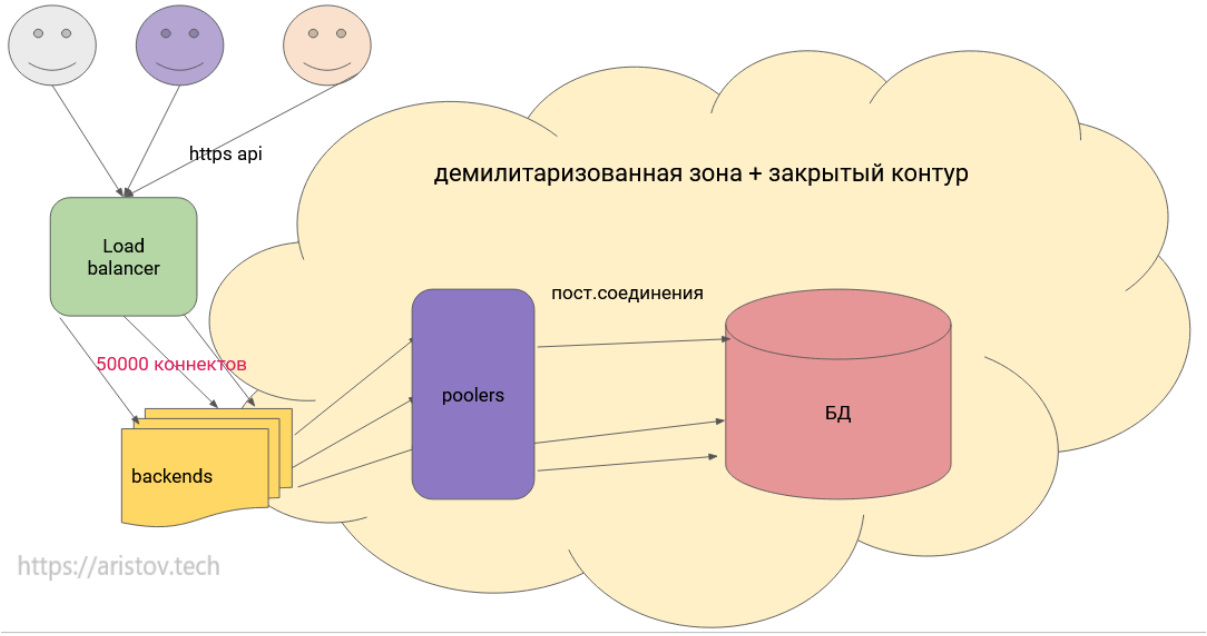
6. Архитектура подключения к PostgreSQL

Вспомним, что для каждого клиента подключение к СУБД представляет собой клон процесса. Он хоть и довольно быстр, но, всё же, это миллисекунды. Кроме этого, выделяется work_mem и temp_buffers на каждый процесс, что тоже несёт свои накладные расходы.

Получается, было бы оптимально не создавать клон процесса и выделять память под каждый запрос, а использовать специальную утилиту, которая как раз и будет держать постоянно висящие подключения и обслуживать посторонние сессии. И такая утилита называется pool connector (pooler или пулер). Также она может помогать терминировать SSL-трафик после демилитаризованной зоны[73] (DMZ).

Что ещё важно в такой схеме подключений – использование балансировщика нагрузки[74], чтобы равномерно распределить нагрузку на бэкенд-процессы.

Схематически можно изобразить архитектуру следующим образом:



Теперь поговорим обо всём этом более подробно и разберём, на что обращать внимание.

[73] DMZ URL: https://club.directum.ru/post/951 (дата обращения 20.03.2026) [1]
[74] балансировка нагрузки URL: https://en.wikipedia.org/wiki/Load_balancing_(computing) (дата обращения 20.03.2026) [2]

В зависимости от задачи и профиля нагрузки обязательно сначала тестируем на лоад-стендах[75] (стенды для нагрузочного тестирования), помня об ограничениях, плюсах и минусах.

На что стоит обратить внимание:

- ❖ количество коннектов
- ❖ нагрузка на сеть/процессор
- ❖ минимизация обмена данными по сети:
  - ➢ делать меньше запросов
  - ➢ читать меньше данных – забываем про select *
  - ➢ упаковываем данные (несколько update insert в одну транзакцию)
  - ➢ уменьшать обработку на лету – используем хранимые процедуры
  - ➢ проанализировать передачу данных:
    - ■ сколько данных было просканировано – сколько отослано
    - ■ сколько было отослано – сколько использовано приложением

Несмотря на встроенный firewall[76] – (настройка **pg_hba и listener)**, PostgreSQL – не средство защиты от DDoS[77] и других атак:

- ❖ желательно работать только в доверенной зоне – ни в коем случае не открывать в свободный интернет
- ❖ меняем стандартный порт
- ❖ аккуратнее поднимаем докер-контейнеры с портом наружу -p 5432:5432 – лучше докер-сеть и по ней доступ к бэкендам (касается и k8s[78] и port-forward[79])
- ❖ сложные пароли – ибо никто ботов с брутфорсом[80] и сканирование портов не отменял

Чек-лист для решений на базе docker:

- ❖ безопасность наше всё!
- ❖ только проверенные источники — официальные образы, bitnami, etc[81]

---

[75] стенды для тестирования URL: https://habr.com/ru/companies/rtlabs/articles/577580/ (дата обращения 20.03.2026) [3]
[76] firewall URL: https://en.wikipedia.org/wiki/Firewall_(computing) (дата обращения 20.03.2026) [4]
[77] DDoS URL: https://selectel.ru/blog/ddos-attacks/ (дата обращения 20.03.2026) [5]
[78] kubernetes
[79] port-forward URL: https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/ (дата обращения 20.03.2026) [6]
[80] брутфорс URL: https://www.anti-malware.ru/threats/brute-force (дата обращения 20.03.2026) [7]
[81] Et cetera – и так далее

- тестируем контейнеры на безопасность[82]
- открывать доступ извне для БД – плохая идея (помним про порты)
- мультистейдж[83]-сборка – в одном окружении собрали jar, потом в минимальном окружении запустили, например, в alpine
- группируем команды, возможно, есть смысл к одному слою всё схлопнуть[84]
- конфиги в каталоге, который и монтируем внутрь контейнера
- указываем версию контейнера – latest плохая практика из-за несовместимости файлов БД разных версий, при рестарте может получиться БД следующей версии, а файлы данных старой версии
- docker теряет популярность из-за отсутствия поддержки c-groups 2.0[85], есть смысл смотреть в сторону containerd и CRI-O[86]

В качестве load balancer (балансировщика нагрузки) на нашей схеме можно использовать или предлагаемый в различных облачных платформах (GCP, AWS, ЯндексОблако и другие), или, например, NGINX[87], или HAproxy[88]. Также дальше будут описаны более сложные инструменты, выполняющие балансировку, как одну из своих функций.

Рассмотрим классические варианты пулеров:

- PgPool II[89]
- pg_bouncer[90] + haproxy
- Odyssey[91]
- ряд готовых кластеров содержит встроенные балансировщики

Характеристики **pgpool II**:

- очень широкий функционал
- используется встроенный механизм репликации
- есть пул соединений

---

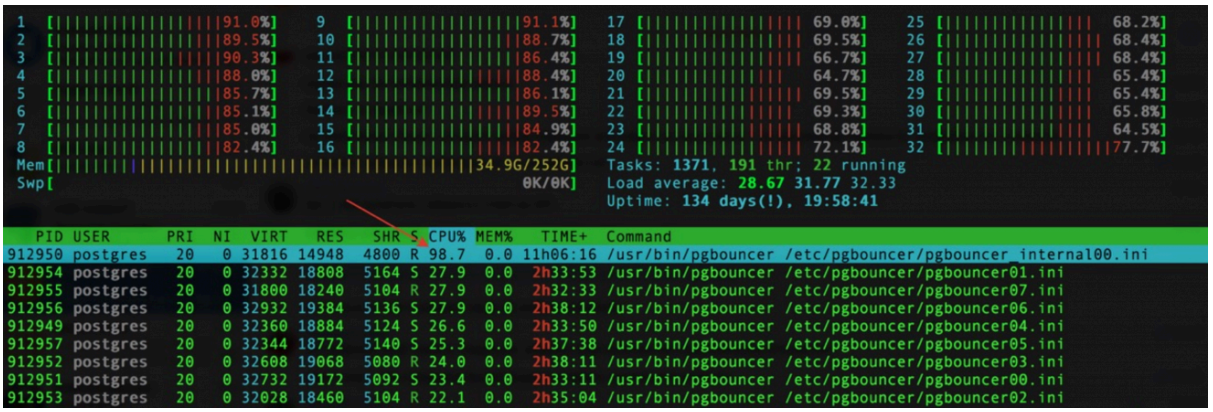
[82] тесты безопасности контейнеров URL: https://habr.com/ru/companies/vk/articles/652149/ (дата обращения 20.03.2026) [8]
[83] multistage URL: https://habr.com/ru/articles/349802/ (дата обращения 20.03.2026) [9]
[84] схлопнуть образ URL: https://habr.com/ru/articles/234829/ (дата обращения 20.03.2026) [10]
[85] c-goups 2.0 URL: https://kubernetes.io/docs/concepts/architecture/cgroups/ (дата обращения 20.03.2026) [11]
[86] containerd URL: https://habr.com/ru/companies/domclick/articles/566224/ (дата обращения 20.03.2026) [12]
[87] NGINX URL: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/ (дата обращения 20.03.2026) [13]
[88] HAproxy URL: https://www.haproxy.org/ (дата обращения 20.03.2026) [14]
[89] pgpool-II URL: https://www.pgpool.net/mediawiki/index.php/Main_Page (дата обращения 20.03.2026) [15]
[90] pgbouncer URL: https://www.pgbouncer.org/ (дата обращения 20.03.2026) [16]
[91] Odyssey URL: https://github.com/yandex/odyssey (дата обращения 20.03.2026) [17]

- ❖ умеет работать на сессию, транзакцию, распараллеливать каждую операцию при условии отсутствия зависимости от изменения данных в той же транзакции
- ❖ балансировщик нагрузки
- ❖ высокая доступность (наблюдатель с виртуальным IP, автоматическое переключение мастера)
- ❖ как раз излишек функционала и есть основная проблема производительности PgPool2[92]

Характеристики **PgBouncer**:

- ❖ только пул соединений
- ❖ легковесный – 2 КБ на соединение
- ❖ можно выбрать тип соединения: на сессию, транзакцию или каждую операцию
- ❖ онлайн-реконфигурация без сброса подключений
- ❖ спроектирован однопоточным
- ❖ сделан максимально простым, в этой простоте масштабируемость не присутствует, как класс
- ❖ сравнение производительности этих двух решений[93]

Основной проблемой PgBouncer является его однопоточность. При этом с недавних пор добавили функционал **reuseport** и теперь можно в несколько копий, но всё равно упираемся в родительский процесс:



[92] pgpool URL: https://severalnines.com/blog/guide-pgpool-postgresql-part-one/ (дата обращения 20.03.2026) [18]

[93] pgpool vs pgbouncer URL: https://scalegrid.io/blog/postgresql-connection-pooling-part-4-pgbouncer-vs-pgpool/ (дата обращения 20.03.2026) [19]

Видим примерно равномерную нагрузку на 32 ядра, 12 дочерних PgBouncer, но больше всего нагружен родительский процесс PgBouncer (вверху), а остальные пулеры ждут его.

Выбирая между этими двумя решениями, нужно ориентироваться на требуемый функционал, однако я рекомендовал бы нагрузочное тестирование в каждом конкретном случае.

На практике может возникать ряд нюансов, например, при переключении с master на secondary автоматическая смена внутри pgbouncer не произойдёт и получится ошибка. Или, при обрыве соединения с пулером, оно продолжит висеть внутри пулера.

Команда Яндекса пыталась решить эти проблемы, но не получилось и пришлось писать свой пулконнектор – Odyssey.

Odyssey не имеет детских болезней, как предыдущие два варианта, но имеет довольно большой порог вхождения для установки и настройки. Про подробную архитектуру Odyssey можно прочитать в статье на Хабр[94].

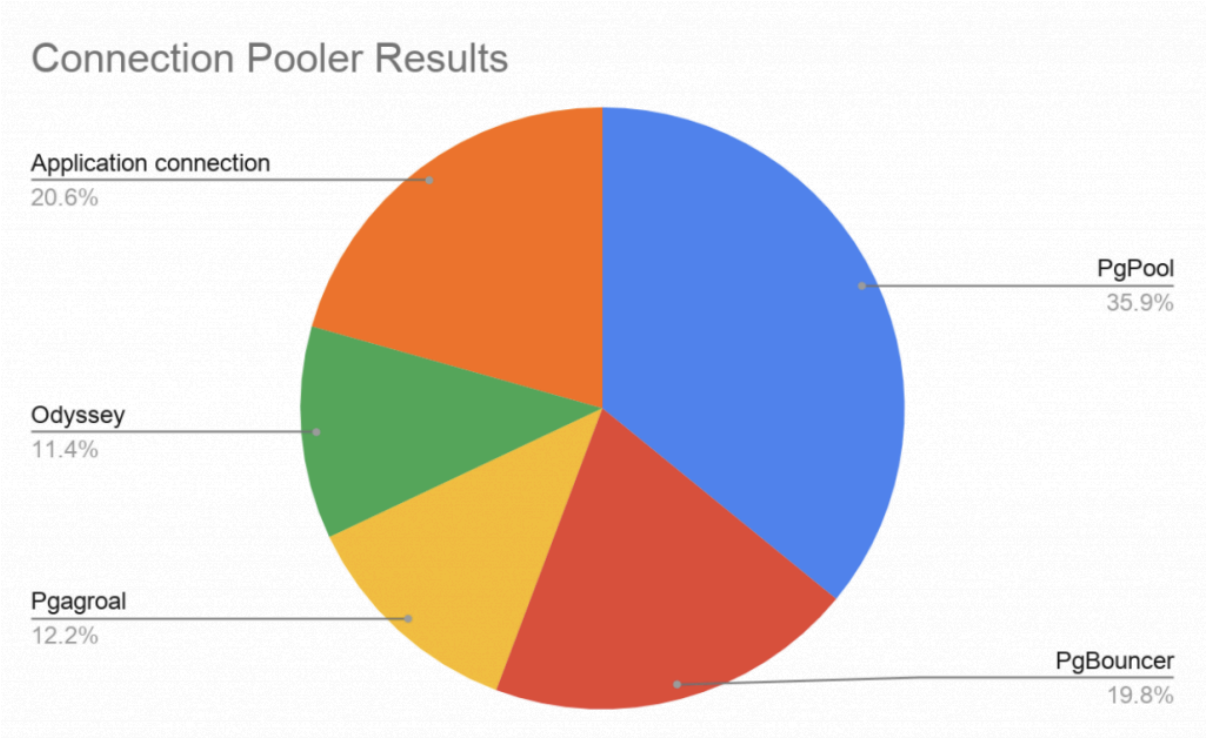
Кроме классических отдельных приложений, у каждого уважающего себя языка программирования есть свой pooler. Пример для Java по ссылке[95]. С ними тоже есть ряд проблем (разбираем на курсе по оптимизации[96]).

В последнее время набирают популярность ещё три решения для пулинга – PgAgroal[97] со множеством отличных фичей и Supavisor[98] под открытой лицензией Apache 2.0[99].

Интересное исследование[100] провели в компании-контрибьюторе[101] PostgreSQL EnterpriseDB[102] по популярности пулеров, опросив более трёх тысяч DBA:

---

[94] Odysey architecture URL: https://habr.com/ru/articles/498250/ (дата обращения 20.03.2026) [20]
[95] Hikari URL: https://www.baeldung.com/java-connection-pooling (дата обращения 20.03.2026) [21]
[96] курс по оптимизации PostgreSQL URL: https://aristov.tech/blog/kurs-po-optimizaczii-postgresql/ (дата обращения 20.03.2026) [22]
[97] pgagroal URL: https://github.com/agroal/pgagroal (дата обращения 20.03.2026) [23]
[98] Supavisor URL: https://github.com/supabase/supavisor (дата обращения 20.03.2026) [24]
[99] Apache 2.0 URL: https://www.apache.org/licenses/LICENSE-2.0 (дата обращения 20.03.2026) [25]
[100] популярность пулеров URL: https://www.enterprisedb.com/blog/what-3000-users-say-about-postgresql-tools-they-use (дата обращения 20.03.2026) [26]
[101] contributors URL: https://www.postgresql.org/community/contributors/ (дата обращения 20.03.2026) [27]
[102] EnterpriseDB URL: https://www.enterprisedb.com/ (дата обращения 20.03.2026) [28]



Видим, что на долю PgBouncer+PgPool приходится больше 55% рынка.

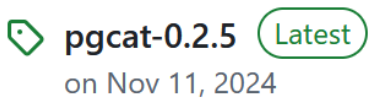
Хотел бы предостеречь от использования стильно-модно-молодёжных новых утилит, которые с огромной вероятностью не дойдут до стабильного релиза 1.0 и будут заброшены. Посмотрим на два примера.

pgcat[103]

Возможности объявлены очень интересные:

- Session mode
- Transaction mode
- Load balancing of read queries
- Failover
- Sharding
- Live configuration reloading
- Mirroring

Но!!! Если посмотрим на дату последнего релиза (на данный момент 06.02.2026):



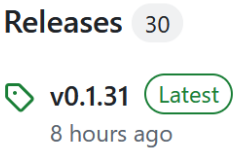
[103] pgcat URL: https://github.com/postgresml/pgcat (дата обращения 16.03.2026) [29]

Соответственно, видим, что разработка остановлена. Старые проблемы не решаются, проект заморожен. *Использование устаревших утилит открывает вектора атак злоумышленникам*.

Следующий проект – **pgdog**.

- ❖ Transaction pooling
- ❖ Load balancer
- ❖ Sharding – должно быть отлично
- ❖ Monitoring

Посмотрим на развитие:



Но! Проекту меньше года, он только набирает популярность и наверняка имеет массу багов и уязвимостей.

В любом случае при применении пулконнекторов или без них рекомендую использовать паттерн circuit breaker[104]. Он позволяет отслеживать неудачно завершившиеся запросы и при этом не просто их заново пытается повторить, тем самым при пиковой нагрузке загоняя СУБД в ещё больший denyall of service, а позволяет, плавно увеличивая интервалы запросов, снизить такую нагрузку. Заодно подойдёт и при ситуациях со сменой лидера и пропавшем коннекте к пулеру.

Важной проблемой является использование SSL (Secure Socket Layer) и терминация шифрованного трафика во внутренней сети для ускорения обработки.

В незащищённой сети использование TLS/SSL обязательно! Также рекомендовано и в демилитаризованной зоне по возможности не отключать шифрование.

Использование SSL для защищённого соединения с PostgreSQL внутри закрытого периметра (например, внутри безопасной сети или сети виртуальных машин) имеет свои плюсы и минусы.

[104] circuit breaker URL : https://java-design-patterns.com/patterns/circuit-breaker/ (дата обращения 20.03.2026) [30]

Плюсы использования SSL:

- **шифрование данных:** SSL обеспечивает шифрование данных между клиентом и сервером, что делает перехват и утечку информации более сложными для злоумышленников, даже если они имеют доступ к внутренней сети
- **доверие и безопасность:** использование SSL помогает подтвердить подлинность сервера перед клиентом и создаёт доверительный канал для обмена данными. Это защищает от атак «человек посередине» (Man-in-the-Middle) и поддерживает целостность данных
- **соответствие стандартам и нормам:** в зависимости от отрасли и законодательства, может потребоваться шифрование данных, даже если они передаются внутри закрытой сети. Использование SSL позволяет соответствовать стандартам безопасности данных
- **защита от внутренних угроз:** внутренние угрозы, такие как злоумышленники внутри сети, могут попытаться перехватить данные или осуществить атаки на базу данных. SSL помогает уменьшить риски таких атак

Минусы использования SSL внутри закрытого периметра:

- **дополнительная нагрузка на производительность:** шифрование и расшифровка данных может вызвать некоторую нагрузку на производительность сервера PostgreSQL. В закрытой сети эта нагрузка может быть излишней
- **сложность настройки:** настройка SSL требует наличия корректных сертификатов, и это может быть сложно в случае внутренних сетей. Неправильная настройка может привести к проблемам соединения
- **увеличение сложности обслуживания:** внедрение SSL увеличивает сложность конфигурации и обслуживания базы данных. Это может потребовать дополнительных шагов при развёртывании и обновлении
- **затраты на ресурсы:** создание и управление сертификатами требует времени и ресурсов. Возможно, потребуется наличие собственной инфраструктуры управления сертификатами

Если посмотреть на приблизительные расходы[105] в виде таблицы:

Результаты тестирования:

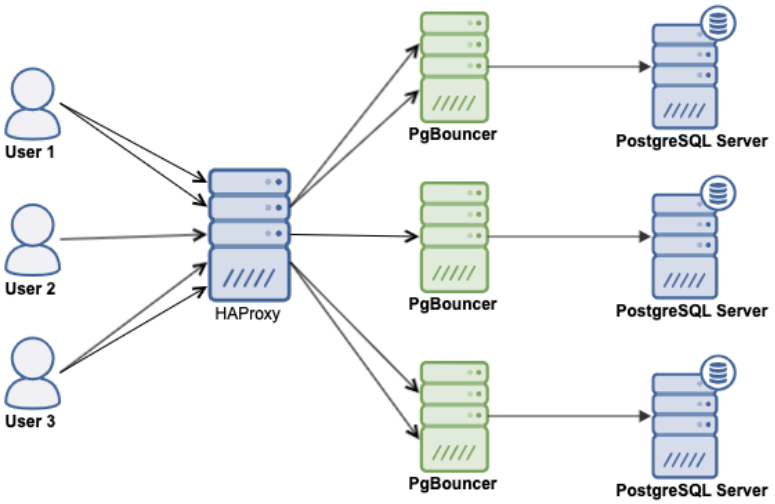
	NO SSL	SSL
Устанавливается соединение при каждой транзакции		
latency average	171.915 ms	187.695 ms
tps including connections establishing	58.168112	53.278062
tps excluding connections establishing	64.084546	58.725846
CPU	24%	28%
Все транзакции выполняются в одно соединение		
latency average	6.722 ms	6.342 ms
tps including connections establishing	1587.657278	1576.792883
tps excluding connections establishing	1588.380574	1577.694766
CPU	17%	21%

В целом, использование SSL внутри закрытого периметра имеет смысл, если стремитесь к обеспечению дополнительного уровня безопасности, защиты данных и соответствия стандартам. Однако следует тщательно взвесить плюсы и минусы в соответствии с требованиями конкретной организации и сетевой инфраструктурой.

Теперь соберём всю информацию вместе и рассмотрим три классических схемы комбинации load balancer + pool connector + PostgreSQL cluster.

[105] расходы на SSL URL: https://habr.com/ru/companies/vk/articles/500708/ (дата обращения 16.03.2026) [31]

Первый вариант:



Пользователи приходят на HAProxy, который выступает, как Load balancer, и знает, где Primary (Master) PostgreSQL node, а где Secondary.

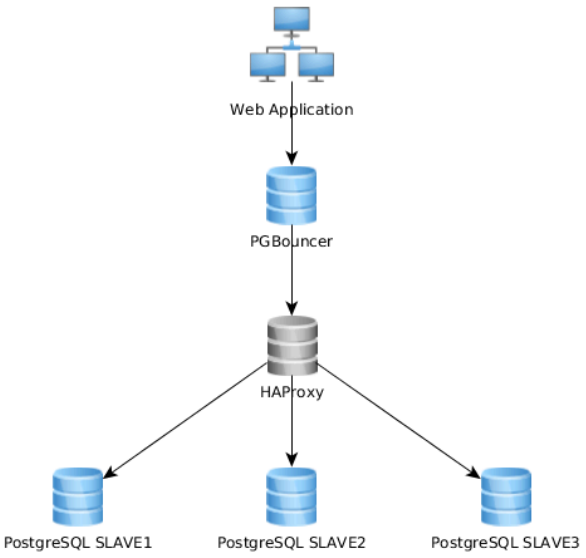
Плюсы:

- на каждый инстанс PostgreSQL свой PgBouncer – решаем проблему однопоточности

Минусы:

- на HAProxy нужно настроить отслеживание статуса ноды PostgreSQL
- нужно настроить равномерную нагрузку на PgBouncer-ноды
- HAProxy – единая точка отказа

Второй вариант:



Пользователи приходят на PgBouncer, который уже идёт на HAProxy, который выступает, как Load balancer, и знает, где Primary (Master) PostgreSQL node, где Secondary.

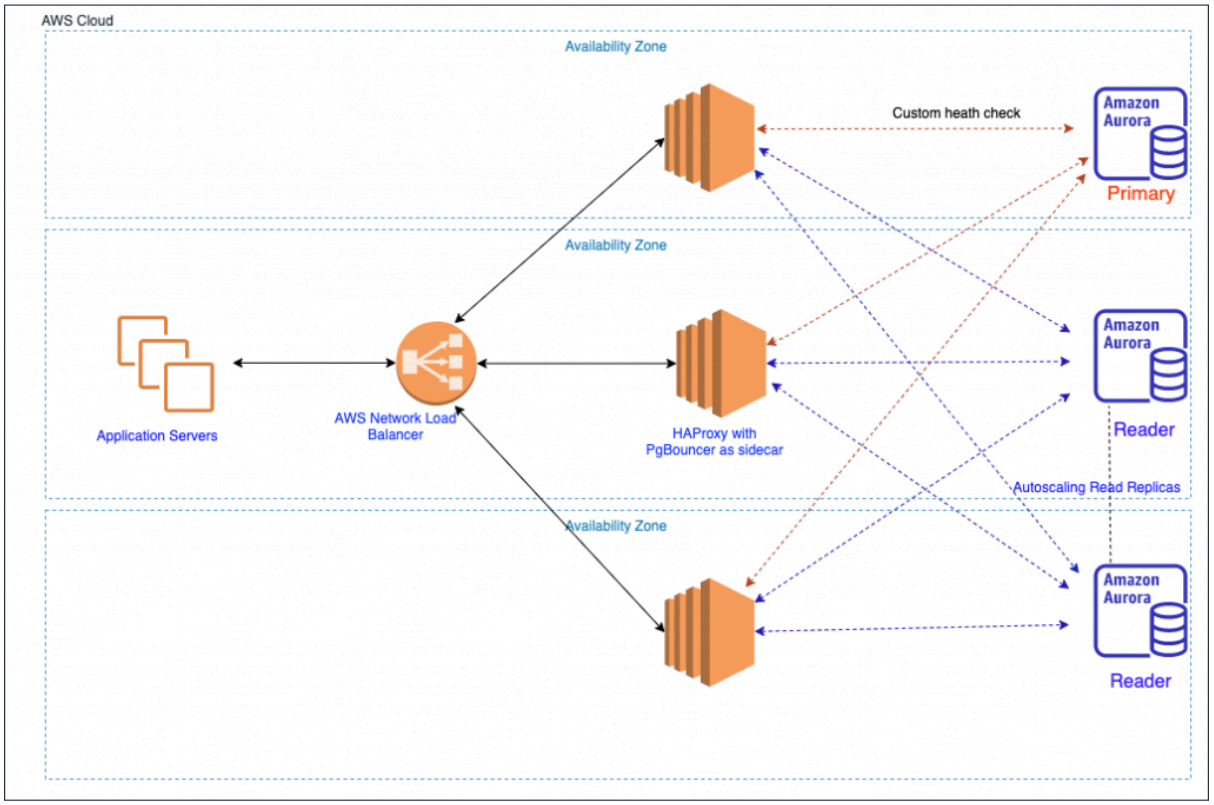
Плюсы:

- простота схемы

Минусы:

- PgBouncer – узкое горлышко из-за однопоточности
- нужно настроить равномерную нагрузку на PostgreSQL-ноды
- как HAProxy, так и PgBouncer – единые точки отказа

Третий вариант (пример из облака Amazon[106]):



Плюсы:

- отличная отказоустойчивость – нет единой точки отказа (облачный load balancer всегда отказоустойчивый)
- схема легко масштабируется
- нет проблемы с однопоточностью PgBouncer – можем добавить дополнительные ноды с HAProxy+PgBouncer

[106] AWS URL: https://aws.amazon.com/ru/ (дата обращения 20.03.2026) [32]

Минусы:

- относительная сложность построения и высокий порог вхождения специалистов
- необходимо немного больше вычислительных ресурсов по сравнению со вторым вариантом

Я рекомендую третий вариант или целиком, или с небольшими изменениями, если вместо облачной инфраструктуры своя on-premise[107].

Важно заметить, что большое влияние может оказать расположение любой утилиты – будь то балансер или пулер – на ноде с PostgreSQL – нет сетевых задержек, но есть аффект на CPU. Всё нужно тестировать! Серебряной пули не существует.

Развернём один из самых популярных пулеров PgBouncer на практике и посмотрим на подводные камни.

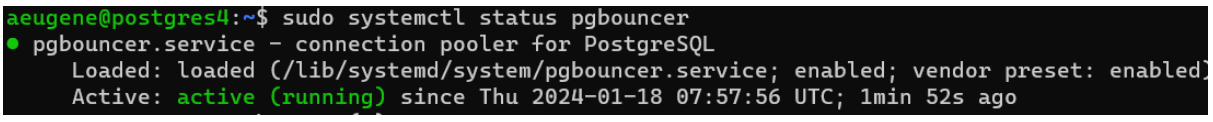
Устанавливать будем на той же ВМ, что и в первой главе. Рекомендую при использовании ВМ в облаке выключать их на время, когда не используете – тогда деньги будут практически не тратиться, только на жёсткий диск и статический IP. Как обычно ни одной правильной и объёмной инструкции в интернете нет, поэтому приступаем.

Установим из пакетного менеджера, повысив свои права до суперадминистратора (sudo), при этом отключим интерактивность вопросов по версиям библиотек (DEBIAN_FRONTEND=noninteractive) и укажем флаг отвечать «**да**» (-y) на вопросы при установке:

```
sudo DEBIAN_FRONTEND=noninteractive apt install -y pgbouncer
```

Проверим статус сервиса после установки:

```
sudo systemctl status pgbouncer
```



Видим статус сервиса Active. Если вывод какой-либо команды открылся в полноэкранном режиме, то для выхода нужно нажать кнопку «q» – quit (выход).

Остановим сервис для внесения изменений в конфиг:

---

[107] on-premise URL: https://www.insight.com/en_US/content-and-resources/glossary/o/on-premises.html (дата обращения 20.03.2026) [33]

```
sudo systemctl stop pgbouncer
```

Далее одной многострочной командой создадим конфигурационный файл, записав строки во временный файл, а потом с правами суперадминистратора перенесём в нужную директорию (иначе не хватит прав). Обратите внимание, что порт у PgBouncer отличается от порта PostgreSQL. Также указываем строку подключения, в данном случае на localhost, и порт PostgreSQL по умолчанию:

```
cat > temp.cfg << EOF
[databases]
thai = host=127.0.0.1 port=5432 dbname=thai
[pgbouncer]
logfile = /var/log/postgresql/pgbouncer.log
pidfile = /var/run/postgresql/pgbouncer.pid
listen_addr = *
listen_port = 6432
auth_type = scram-sha-256
auth_file = /etc/pgbouncer/userlist.txt
admin_users = admindb
EOF
cat temp.cfg | sudo tee -a /etc/pgbouncer/pgbouncer.ini
```

Очень важно обратить внимание на метод шифрования пароля для подключения! С версии 18 у PostgreSQL шифрование по умолчанию в SCRAM-SHA-256[108]. Но в данной ситуации разработчики PgBouncer, к сожалению, реализовали поддержку шифрования для этого стандарта только в конце 2023 года, и в конфигурационном файле `auth_file` необходимо будет указывать пароль в открытом виде (с конца 2023 можно использовать и шифрованный вид). В случае шифрования в md5 – он будет везде зашифрован, но стандарт уже устарел, хоть ещё и возможно его использовать.

Создадим список пользователей для доступа:

```
cat > temp2.cfg << EOF
"admindb" "md5a1edc6f635a68ce9926870fe752e8f2b"
"postgres" "admin123#"
EOF
cat temp2.cfg | sudo tee -a /etc/pgbouncer/userlist.txt
```

Видно, что у md5 указывается шифрованный пароль, для SCRAM-SHA-256 он открытый.

Для SCRAM-SHA-256 в новых версиях это выглядело бы так:

---

[108] MD5 vs SCRAM-SHA-256 URL: https://www.cybertec-postgresql.com/en/from-md5-to-scram-sha-256-in-postgresql/ (дата обращения 20.03.2026) [34]

```
"postgres"
"SCRAM-SHA-256$4096:eM1ToRH8QOewbDddWnxzBQ==$ktPLPRkEPtMr1epwtJv1HxVHAxjsM+
KEbLaW7loiBQs=:UMtetDFOi30NB56aX4JBT3lXudOClkANX02Xxhjjg1U="
```

Шифрованные пароли можно автоматически выгрузить из PostgreSQL командой:

```
sudo -u postgres psql -Atq -h 127.0.0.1 -p 5432 -U postgres -d
postgres -c "SELECT concat('\"', username, '\" \"', passwd, '\"') FROM
pg_shadow" >> /tmp/userlist.txt && sudo mv /tmp/userlist.txt
/etc/pgbouncer/userlist.txt
```

Нам такой вариант подойдёт только в новых версиях из-за новой системы шифрования SCRAM-SHA-256!

Здесь есть важный подводный камень – так как внутри пароля SCRAM используются символы $, то при перенаправлении вывода Linux может их воспринимать, как служебный вызов переменных, и итоговый пароль будет НЕВЕРЕН! Для корректного перенаправления необходимо использовать экранирование – в случае с EOF использовать прямые кавычки 'EOF' вместо EOF. Тогда весь дальнейший ввод будет экранирован.

Итоговый вариант с шифрованием был бы такой:

```
cat > temp2.cfg << 'EOF'
"admindb"
"SCRAM-SHA-256$4096:sx9vFfoPimVB8fpivzgAGQ==$O0n8/0F4GyhRPV24bEGoQuj6
H4zpmZCfT49pLHpZZy4=:BJZ64nq3G/rN/pX1eWnLzp0iSEMFsIF3RRPmvmmgOp0="
"postgres"
"SCRAM-SHA-256$4096:M1PRI7iJscoRk+A9lp4srQ==$CBS/7nmUo6Q/CkU93bk8rl3A
MrESgn0lOtY83dPE2VY=:PkX4G6QOvs1F1bTrQ9I4/kKIyRn6EP8Hz+iIrf5ptTo="
EOF
cat temp2.cfg | sudo tee /etc/pgbouncer/userlist.txt
```

Зададим пароль пользователю postgres (будет в SCRAM, так как он задан по умолчанию в системе) и создадим пользователя admindb, указав сразу шифрованный в md5 пароль (начинается с md5):

```
sudo -u postgres psql -c "ALTER USER postgres WITH PASSWORD
'admin123#';"
sudo -u postgres psql -c "create user admindb with password
'md5a1edc6f635a68ce9926870fe752e8f2b';"
```

Обратите внимание, что прямо из командной строки выполнили команды в PostgreSQL, при этом сделали под простым Linux-пользователем с правами на sudo.

Обратите внимание – при использовании хэшей SCRAM они должны совпадать и в PostgreSQL, и в pgbouncer.

Чтобы не вводить пароль каждый раз при использовании нашего пользователя, создадим файл .pgpass в домашней директории пользователя postgres:

```
echo "localhost:5432:thai:postgres:admin123#" | sudo tee -a /var/lib/postgresql/.pgpass && sudo chmod 600 /var/lib/postgresql/.pgpass
```

К сожалению, PostgreSQL пока не поддерживает шифрование этого файла.

Далее включаем listener на прослушивание сетевого трафика, а не только localhost, и задаём маску подсети для доступа (10.* – это внутренняя подсеть, не интернет):

```
echo "listen_addresses = '*'" >> /etc/postgresql/18/main/postgresql.conf
echo "host all all 10.0.0.0/8 scram-sha-256" | sudo tee -a /etc/postgresql/18/main/pg_hba.conf
sudo pg_ctlcluster 18 main restart
```

Обратите внимание, что при наличии двух строк в файле pg_hba.conf, отвечающего за настройки firewall, для которых подходит наш IP-адрес, **приоритет отдаётся строке** с новым методом шифрования **SCRAM-SHA-256**!

Включаем автозагрузку сервиса PgBouncer и стартуем его:

```
sudo systemctl enable pgbouncer
sudo systemctl start pgbouncer
```

Попытаемся получить доступ к нашему кластеру PostgreSQL через PgBouncer (порт 6432):

```
sudo -u postgres psql -p 6432 -h 127.0.0.1 -d thai -U postgres
```

```
aeugene@postgres4:~$ sudo -u postgres psql -p 6432 -h 127.0.0.1 -d thai -U postgres
Password for user postgres:
psql (16.1 (Ubuntu 16.1-1.pgdg22.04+1))
Type "help" for help.

thai=#
```

Если что-то пошло не так, то можно посмотреть настройки, логи или рестартовать PgBouncer (утилита tail[109] показывает конец файла):

```
tail /var/log/postgresql/postgresql-18-main.log
tail /var/log/postgresql/pgbouncer.log
sudo cat /etc/pgbouncer/pgbouncer.ini
sudo cat /etc/pgbouncer/userlist.txt
sudo cat /etc/postgresql/18/main/pg_hba.conf
sudo cat /etc/postgresql/18/main/postgresql.conf
sudo systemctl restart pgbouncer
```

Sudo используется для повышения прав моего пользователя aeugene. Также альтернатива – переключиться под Linux-пользователя postgres и работать под ним для доступа к файлам PostgreSQL и PgBouncer.

Для доступа к настройкам PgBouncer нужно указать при подключении БД pgbouncer и пользователя admindb, которого создали чуть ранее и прописали, как администратора в конфигурационном файле:

```
sudo -u postgres psql -p 6432 -h 127.0.0.1 -d pgbouncer -U admindb
```

```
aeugene@postgres4:~$ sudo -u postgres psql -p 6432 -h 127.0.0.1 -d pgbouncer -U admindb
Password for user admindb:
psql: error: connection to server at "127.0.0.1", port 6432 failed: FATAL:  SASL authentication failed
```

Получили ошибку, потому что у пользователя метод шифрования md5.

Здесь есть два варианта:

- для доступа по md5 нужно установить метод шифрования в PostgreSQL и внести корректировки в firewall – **pg_hba.conf**
- использовать новый метод шифрования, при этом не забыв поменять на открытый пароль строку в файле userlist, в новой версии можно шифрованную строку

Сравним скорости работы, используя наши профили нагрузок.

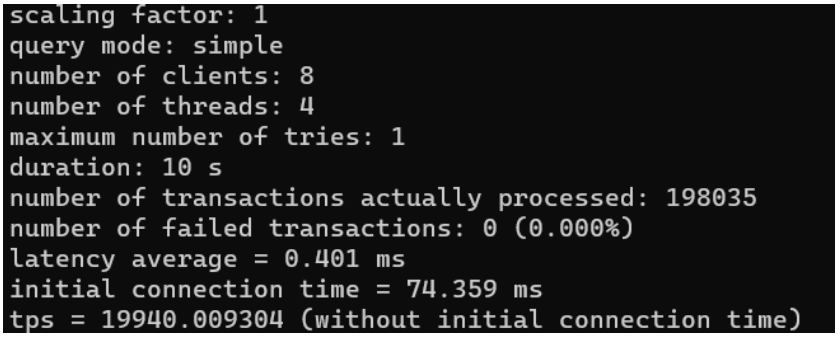
Первым сравним чтение. Перейдя сначала под Linux-пользователя postgres (порт 5432) и используя утилиту нагрузочного тестирования **pgbench**[110] с кастомным скриптом workload.sq проверим количество отработавших запросов:

```
sudo su postgres
cat > ~/workload.sql << EOL
```

[109] tail URL: https://www.linuxfoundation.org/blog/blog/classic-sysadmin-14-tail-and-head-commands-in-linux-unix (дата обращения 20.03.2026) [35]

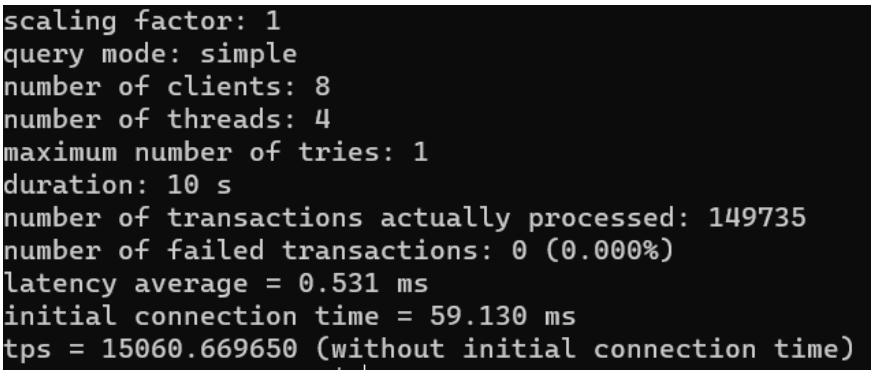
[110] pgbench URL: https://www.postgresql.org/docs/current/pgbench.html (дата обращения 16.03.2026) [36]

```
\set r random(1, 5000000)
SELECT id, fkRide, fio, contact, fkSeat FROM book.tickets WHERE id =
:r;
EOL
/usr/lib/postgresql/18/bin/pgbench -c 8 -j 4 -T 10 -f ~/workload.sql
-U postgres -p 5432 -h 127.0.0.1 thai
```



Второй тест, используя pgbouncer (порт 6432):

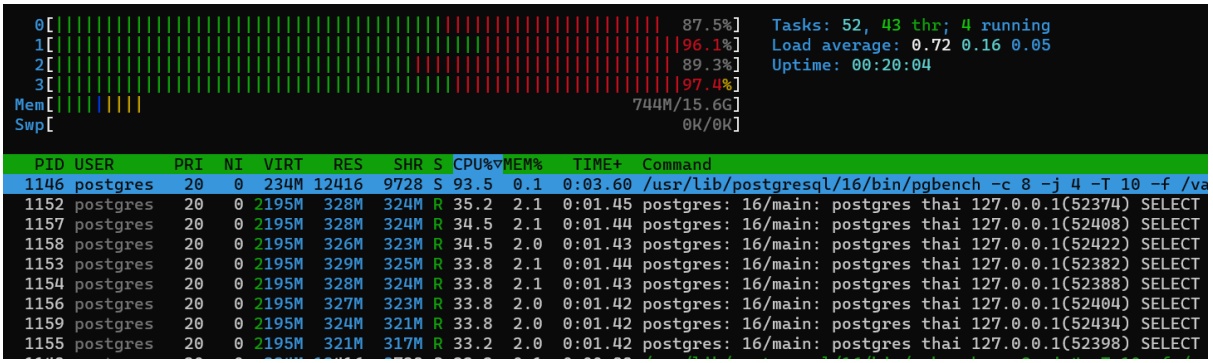
```
/usr/lib/postgresql/18/bin/pgbench -c 8 -j 4 -T 10 -f
~/workload.sql -U postgres -p 6432 -h 127.0.0.1 thai
```



Видим падение производительности ~25%.

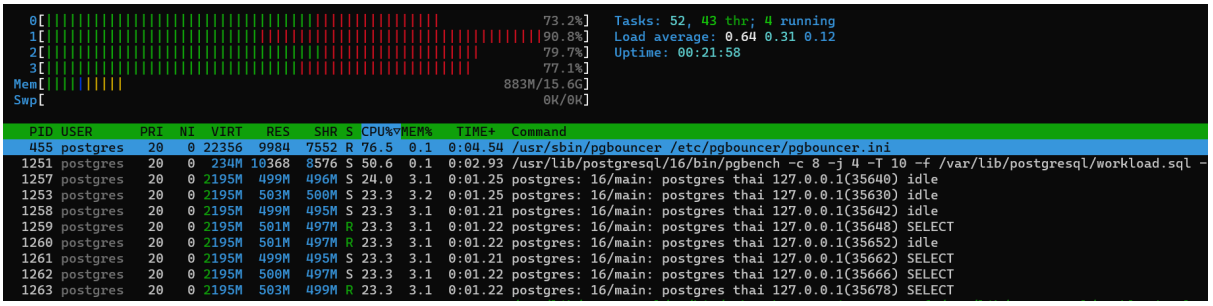
Чтобы понять, что происходит, посмотрим на нагрузку ВМ.

В первом случае:



Полная утилизация всех четырёх ядер.

В случае использования pg_bouncer:



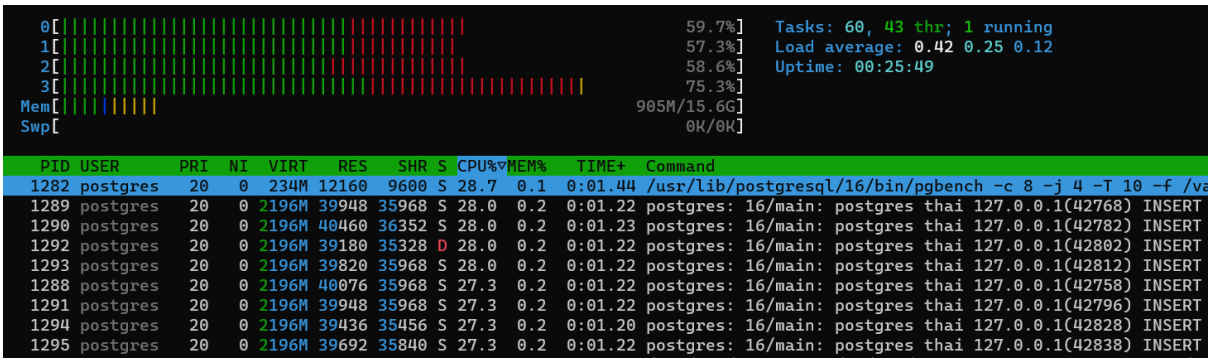
Видим 100% использование одного ядра под pgbouncer и не сильно высокую нагрузку на другие ядра, и даже idle висящих процессов PostgreSQL – **упираемся в одноядерность pgbouncer**!

Важно посмотреть не только профиль на чтение данных, а как аффектит пулер на операции записи. Используем профиль генерации случайных записей в таблицу перевозок (сценарий workload2):

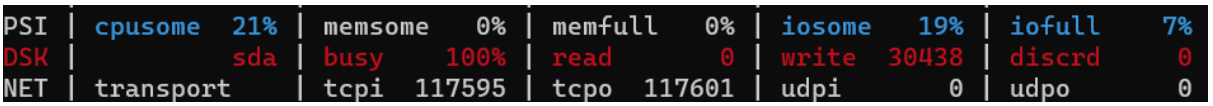
```
cat > ~/workload2.sql << EOL
INSERT INTO book.tickets (fkRide, fio, contact, fkSeat)
VALUES (
    ceil(random()*100)
     , (array(SELECT fam FROM book.fam))[ceil(random()*110)]::text ||
' ' ||
    (array(SELECT nam FROM book.nam))[ceil(random()*110)]::text
                  ,('{"phone":"+7'     ||     (1000000000::bigint     +
floor(random()*9000000000)::bigint)::text || '"}')::jsonb
    , ceil(random()*100));
EOL
/usr/lib/postgresql/18/bin/pgbench -c 8 -j 4 -T 10 -f ~/workload2.sql
-U postgres -p 5432 -h 127.0.0.1 thai
```

```
number of clients: 8
number of threads: 4
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 61565
number of failed transactions: 0 (0.000%)
latency average = 1.291 ms
initial connection time = 73.554 ms
tps = 6195.589835 (without initial connection time)
```

Нагрузка на CPU (htop):



Нагрузка на диск (atop):



Ожидаемо, процессор простаивает, а диск 100% нагружен – упёрлись в диск. Так как pgbouncer чувствителен именно к процессору, сделаем предположение, что он как раз потратит свободное процессорное время и не будет падения производительности:

`/usr/lib/postgresql/18/bin/pgbench -c 8 -j 4 -T 10 -f ~/workload2.sql -U postgres -p 6432 -h 127.0.0.1 thai`

```
number of clients: 8
number of threads: 4
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 54401
number of failed transactions: 0 (0.000%)
latency average = 1.463 ms
initial connection time = 56.054 ms
tps = 5469.614587 (without initial connection time)
```

Всё-таки есть небольшая просадка ~10%.

Обратите внимание, что в реальных условиях, особенно при сетевом варианте использования пулеров, просадка может быть более ощутимой за счёт большего количества сетевых прыжков.

Теперь, чтобы зайти в конфиг pgbouncer, поменяем метод шифрования на md5 и, заодно, протестируем скорость.

Необходимо включить md5 в PostgreSQL:

```
psql -c "ALTER SYSTEM SET password_encryption = 'md5';"
```

Изменить в файле pg_hba.conf значение scram-sha-256 на md5:

```
nano /etc/postgresql/18/main/pg_hba.conf
```

Применить изменения в конфигурации:

```
psql -c "SELECT pg_reload_conf();"
```

Изменить пароль пользователя postgres в PostgreSQL, чтобы он стал зашифрован md5:

```
psql -c "ALTER USER admindb WITH PASSWORD 'admin123#';";
```

Изменить в файле pgbouncer.ini значение scram-sha-256 на md5 (возможно, потребуется пользователь с правами sudo):

```
nano /etc/pgbouncer/pgbouncer.ini
```

Достать новый пароль в md5 пользователя postgres из PostgreSQL:

```
psql -c "SELECT usename,passwd from pg_shadow;"
```

Изменить пароль пользователя postgres в pgbouncer, чтобы он стал зашифрован md5:

```
nano /etc/pgbouncer/userlist.txt
```

Рестартуем pgbouncer для перечитывания изменений (требуется пользователь с правами sudo):

```
sudo systemctl restart pgbouncer
```

Необходимо отметить, что для применения изменённых значений pgbouncer можно рестартовать через systemctl, но также у него имеется ключ «-r» для онлайн-реконфигурации.

При коннекте указываем пользователя и какие у него есть права на те или иные действия.

Замерим скорость чтения с шифрованием md5:

```
number of clients: 8
number of threads: 4
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 242156
number of failed transactions: 0 (0.000%)
latency average = 0.327 ms
initial connection time = 97.385 ms
tps = 24452.330346 (without initial connection time)
```

Вот такой сюрприз – с md5 скорость выше, так как используется более легковесный метод шифрования.

Давайте разберёмся, почему так.

Начнём с протокола установки соединения:

Метод	Этапы	Round-trips	Комментарий
`trust`	0	0	Не аутентифицирует — моментально.
`md5`	1	1	Клиент → сервер: `md5(password + username)`
`scram-sha-256`	3	**3**	Требует полноценного SASL-диалога: `ClientHello` → `ServerHello+Nonce` → `ClientFinal`

Диалог SCRAM (упрощённо):
C → S: AuthenticationSASL (SCRAM-SHA-256)
S → C: AuthenticationSASLContinue + nonce (случайная строка)
C → S: AuthenticationSASLResponse — клиент вычисляет ClientProof, отправляет c=...,r=...,p=...
S → C: AuthenticationSASLFinal — сервер проверяет proof и подтверждает
⚠️ Каждый шаг — это один сетевой round-trip (RTT).
На локальной машине RTT ~0.1 мс, но по сети (например, из облака в облако, или через NAT/VPC) — легко 1–10 мс на RTT, итого +2–20 мс к initial connect.

Но всё же безопасность важнее. Если у вас закрытый контур и происходят постоянные подключения к PostgreSQL, то для ускорения работы, можно понизить уровень безопасности до md5.

В этой главе рассмотрели архитектуру подключения в PostgreSQL. Увидели варианты выбора пулеров и настроили pgbouncer. В следующей главе рассмотрим работу с правами пользователя.

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/06/connect.sql

7. Права пользователей, RLS

Роль[111] – сущность, которая может владеть объектами и иметь определённые права в базе на доступ к другим объектам. Роль может представлять «пользователя», «группу» или и то и другое в зависимости от варианта использования.

Роли/пользователи базы данных полностью отличаются от пользователей операционной системы. На практике поддержание соответствия между ними может быть удобным, но не является обязательным.

Роли базы данных являются глобальными для всей установки кластера базы данных, а не для отдельной базы данных.

Для создания роли используется команда SQL `CREATE ROLE name;` `CREATE GROUP name;` или `CREATE USER name;` Команды идентичны, за исключением того, что в последнем случае задастся ещё параметр **LOGIN** (чуть далее более подробно рассмотрим этот параметр).

Для удаления роли используется команда `DROP ROLE имя;`

Для модификации – `ALTER ROLE имя;`

Для начальной настройки кластера базы данных система сразу после инициализации обычно содержит одну роль **postgres** и эта роль является **суперпользователем.** Для создания других ролей вначале нужно подключиться с этой ролью.

Каждое подключение к серверу базы данных выполняется под именем конкретной роли и эта роль определяет начальные права доступа для команд, выполняемых в этом соединении. Например, программа **psql** для указания роли использует аргумент командной строки **-U**.

Необходимо упомянуть псевдороль **public**, которая **неявно включает** в себя **все** остальные роли.

О ней пример чуть позже на практике.

---

[111] роль URL: https://www.postgresql.org/docs/18/user-manag.html (дата обращения 19.03.2026) [1]

Рассмотрим синтаксис команды[112]:
CREATE ROLE name [[**WITH**] parameter [...]]

Здесь можно задать от нуля и более параметров. Давайте разбираться.

Более подробно (**жирным** значение по умолчанию):

- **name** – имя создаваемой роли
- SUPERUSER **/ NOSUPERUSER** – будет ли эта роль «суперпользователем», для которого нет ограничений доступа в базе данных, за небольшим исключением при выполнении функций с параметром права создавшего (в рамках этой книги не рассматривается). Статус суперпользователя несёт опасность и назначать его следует только в случае необходимости. Создать нового суперпользователя может только суперпользователь
- CREATEDB **/ NOCREATEDB** – может ли роль создавать базы данных. Указание CREATEDB даёт новой роли это право, а NOCREATEDB запрещает роли создавать базы данных
- CREATEROLE **/ NOCREATEROLE** – может ли роль создавать новые роли. Роль с правом CREATEROLE может также изменять и удалять другие роли
- **INHERIT /** NOINHERIT – будет ли роль «наследовать» права ролей, членом которых она является
- LOGIN **/ NOLOGIN** – разрешается ли новой роли вход на сервер; то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Принято считать, что роль с атрибутом LOGIN соответствует пользователю. Роли/пользователи без этого атрибута считаются группами/ролями. При вызове CREATE USER – LOGIN добавится по умолчанию
- REPLICATION **/ NOREPLICATION** – роль/пользователь для управления репликацией
- BYPASSRLS **/ NOBYPASSRLS** – будут ли для роли игнорироваться все политики защиты на уровне строк (RLS – row level security). Создавать роли с атрибутом BYPASSRLS разрешено только суперпользователям

Необходимо помнить, что **pg_dump** (утилита для бэкапа, будет рассмотрена в главе про резервное копирование) по умолчанию отключает **row_security** (устанавливает значение OFF), чтобы гарантированно было выгружено всё содержимое таблицы. Если пользователь, запускающий **pg_dump**, не будет иметь необходимых прав, он получит ошибку. На

[112] create role URL: https://www.postgresql.org/docs/18/sql-createrole.html (дата обращения 19.03.2026) [2]

суперпользователей и владельцев требуемых БД/таблиц это не распространяется.

- **CONNECTION LIMIT** – устанавливаем предел подключений. Если роли разрешён вход, этот параметр определяет, сколько параллельных подключений может установить роль. Default -1 также снимает это ограничение
- [ENCRYPTED] PASSWORD 'password' **/ PASSWORD NULL** – задаёт пароль роли с атрибутом LOGIN. Для групп/ролей пароль не нужен. Шифрование в версии 18 по умолчанию **scram-sha-256**
- **VALID UNTIL** 'timestamp' – устанавливает дату и время, после которого пароль роли перестаёт действовать. Если это предложение отсутствует, срок действия пароля будет неограниченным
- **IN ROLE** role_name – перечисляются одна или несколько существующих ролей, в которые будет немедленно включена новая роль. При этом добавить новую роль с правами администратора таким образом нельзя, для этого надо отдельно выполнить команду выдачи привилегий GRANT (чуть позже рассмотрим в этой главе)
- **ROLE** role_name – перечисляются одна или несколько существующих ролей, которые автоматически становятся членами создаваемой роли (по сути, таким образом новая роль становится «группой»)

Посмотрим на практике.

Для получения списка существующих ролей, в том числе системных, используем команду pg_roles:

```
SELECT rolname FROM pg_roles;
```

Для получения списка актуальных ролей/пользователей:

```
SELECT usename, usesuper FROM pg_catalog.pg_user;
```

Метакоманда `\du` программы **psql** также полезна для получения списка существующих ролей, заодно посмотрим и выданные права.

```
postgres=# SELECT rolname FROM pg_roles;
          rolname
---------------------------
 pg_monitor
 pg_read_all_settings
 pg_read_all_stats
 pg_stat_scan_tables
 pg_read_server_files
 pg_write_server_files
 pg_execute_server_program
 pg_signal_backend
 postgres
(9 rows)

postgres=# SELECT usename, usesuper FROM pg_catalog.pg_user;
 usename  | usesuper
----------+----------
 postgres | t
(1 row)

postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Создадим роль:

```
CREATE ROLE test;
```

Посмотрим на неё и её права:

```
SELECT * FROM pg_catalog.pg_user;
```

Или вторым способом:

```
\du
```

```
postgres=# CREATE ROLE test;
CREATE ROLE
postgres=# SELECT * FROM pg_catalog.pg_user;
 usename  | usesysid | usecreatedb | usesuper | userepl | usebypassrls |  passwd  | valuntil | useconfig
----------+----------+-------------+----------+---------+--------------+----------+----------+-----------
 postgres |       10 | t           | t        | t       | t            | ******** |          |
(1 row)

postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 test      | Cannot login                                               | {}
```

Видим, что PostgreSQL не считает эту роль пользователем, так как у неё нет права на логин (**cannot login**).

Дадим роли логин:

```
ALTER USER test LOGIN;
```

Попробуем законнектиться:

```
\c - test
```

```
postgres=# ALTER USER test LOGIN;
ALTER ROLE
postgres=# \c - test
FATAL:  Peer authentication failed for user "test"
Previous connection kept
```

И видим ошибку.

Это произошло по той причине, что не меняли базовые настройки PostgreSQL, а по умолчанию разрешён вход только доверенным Linux-пользователям через **unix socket**[113]. Зайдём в Linux под пользователем postgres и запустим psql – PostgreSQL увидел, что зашли под авторизованным пользователем Linux и авторизовал своего пользователя postgres.

Для включения входа по паролю необходимо поменять настройку в файле **pg_hba.conf** и включить вход по паролю с шифрованием **scram-sha-256**[114] (версия 14+) / **MD5**[115] (версия 13 и ниже):

```
nano /etc/postgresql/18/main/pg_hba.conf
peer -> scram-sha-256
```

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     scram-sha-256
```

После изменений в системных файлах необходимо перезапустить кластер под root или пользователя с доступом к sudo:

```
pg_ctlcluster 18 main restart
```

Зададим пароль пользователю test:

```
\password test
```

Второй вариант:

```
ALTER USER test PASSWORD 'pass$123';
```

---

[113] unix socket URL: https://lecturesnet.readthedocs.io/net/low-level/ipc/socket/unix.html (дата обращения 19.03.2026) [3]
[114] scram-sha-256 URL: https://en.wikipedia.org/wiki/Salted_Challenge_Response_Authentication_Mechanism (дата обращения 19.03.2026) [4]
[115] MD5 URL: https://en.wikipedia.org/wiki/MD5 (дата обращения 19.03.2026) [5]

Попробуем законнектиться:
`\c - test`

Получим список объектов:
`\dt`

Получим доступ к данным:
`SELECT * FROM t;`

```
postgres=# ALTER USER test PASSWORD 'pass$123';
ALTER ROLE
postgres=# \c - test
Password for user test:
You are now connected to database "postgres" as user "test".
postgres=> \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | t    | table | postgres
(1 row)

postgres=> select * from t;
ERROR:  permission denied for table t
```

При этом получили список объектов только в схеме public – по умолчанию получаем туда права, но **к данным в чужом объекте доступ ожидаемо не получили**.

Также кроме общих прав на подключение, создание БД и остальных, есть и более тонкая настройка на все объекты БД – привилегии. Можно как их выдать **GRANT**[116], так и отозвать **REVOKE**. Тема очень огромная по тонкой настройке прав доступа, затронем её основные моменты.

Команда **GRANT** имеет две основные разновидности: первая назначает права для доступа к объектам баз данных (таблицам, столбцам, представлениям, сторонним таблицам, последовательностям, базам данных, функциям, процедурам, схемам или табличным пространствам и т.п.), а вторая назначает одни роли членами других. Эти разновидности во многом похожи, но имеют достаточно отличий, чтобы рассматривать их отдельно.

GRANT для объектов баз данных

[116] GRANT URL: https://www.postgresql.org/docs/18/sql-grant.html (дата обращения 19.03.2026) [6]

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
TRIGGER }
[, ...] | ALL [PRIVILEGES] }
ON { [TABLE] table_name [, ...]
| ALL TABLES IN SCHEMA schema_name [, ...] }
TO role_specification [, ...] [WITH GRANT OPTION]

where role_specification can be:

[GROUP] role_name
| PUBLIC
| CURRENT_ROLE
| CURRENT_USER
| SESSION_USER

Эта разновидность команды GRANT даёт одной или нескольким ролям определённые права для доступа к объекту базы данных (таблицы, функции, процедуры, внешние таблицы и так далее). Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами.

Ключевое слово PUBLIC означает, что права даются всем ролям, включая те, что могут быть созданы позже. PUBLIC можно воспринимать, как неявно определённую группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли PUBLIC.

Если указано WITH GRANT OPTION, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет. Группе PUBLIC право передачи права дать нельзя.

Нет необходимости явно давать права для доступа к объекту его владельцу (обычно это пользователь, создавший объект), так как по умолчанию он имеет все права (однако владелец может лишить себя прав в целях безопасности).

Право удалять объект или изменять его определение произвольным образом не считается назначаемым; оно неотъемлемо связано с владельцем, так что отозвать это право или дать его кому-то другому нельзя. Владелец также неявно получает право распоряжения всеми правами для своего объекта.

Возможные права:

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**
- **TRUNCATE**
- **REFERENCES**
- **TRIGGER**
- **CREATE**
- **CONNECT**
- **TEMPORARY**
- **EXECUTE**
- **USAGE**

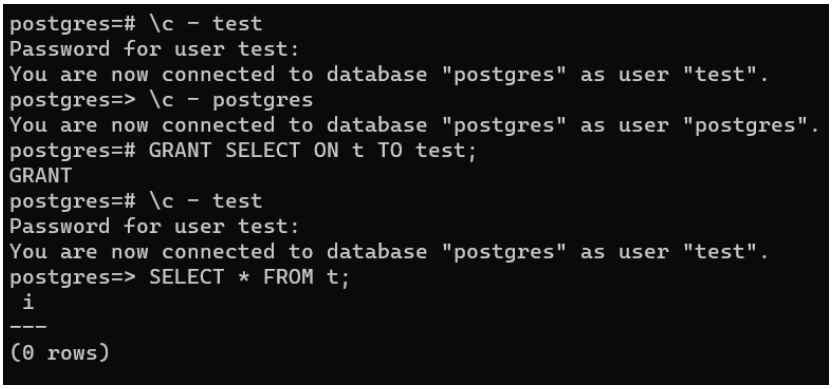
Можно сразу **ALL PRIVILEGES** – даёт целевой роли все права, применимые к данному объекту. Ключевое слово PRIVILEGES является необязательным в PostgreSQL, хотя в стандарте SQL оно требуется.

Также помните про владельца объекта – роль, создавшая объект. По умолчанию имеет полные права на объект. Также права имеют роли, включённые в неё. Владелец может быть изменён командой ALTER ... OWNER TO роль.

Посмотрим на практике.

Зайдём под суперпользователем и дадим права на выборку из таблицы t нашему пользователю test:

```
\c - postgres
GRANT SELECT ON t TO test;
\c - test
SELECT * FROM t;
```



Данные из таблицы t успешно получили.

GRANT для ролей

GRANT role_name [, ...] **TO** role_specification [, ...]
[WITH ADMIN OPTION]
[GRANTED BY role_specification]

Эта разновидность команды GRANT включает роль в члены одной или нескольких других ролей. Членство в ролях играет важную роль, так как права, данные роли, распространяются и на всех её членов.

Получивший членство в роли с указанием WITH ADMIN OPTION сможет, в свою очередь, включать в члены этой роли, а также исключать из неё, другие роли.

Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Роли с правом **CREATEROLE** могут управлять членством в любых ролях, кроме ролей суперпользователей.

Посмотреть права очень просто:
\dp t → **test=r**
имя_роли=xxxx – права, назначенные роли
=xxxx – права, назначенные PUBLIC
r – SELECT ("read", чтение)
w – UPDATE ("write", запись)
a – INSERT ("append", добавление)
d – DELETE
D – TRUNCATE
x – REFERENCES
t – TRIGGER
X – EXECUTE
U – USAGE
C – CREATE
c – CONNECT
T – TEMPORARY

arwdDxt – ALL PRIVILEGES (все права для таблиц; для других объектов другие)
***** – право передачи заданного права
/yyyy – роль, назначившая это право

```
postgres=> \dp t
                              Access privileges
 Schema | Name | Type  |     Access privileges     | Column privileges | Policies
--------+------+-------+---------------------------+-------------------+----------
 public | t    | table | postgres=arwdDxt/postgres+|                   |
        |      |       | test=r/postgres           |                   |
(1 row)
```

REVOKE[117]

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
      TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] table_name [, ...]
     | ALL TABLES IN SCHEMA schema_name [, ...] }
  FROM role_specification [, ...]
  [ CASCADE | RESTRICT ]
```

Команда REVOKE лишает одну или несколько ролей прав, назначенных ранее. Ключевое слово PUBLIC обозначает **неявно** определённую группу всех ролей.

Любая конкретная роль получает в сумме права, данные непосредственно ей, права, данные любой роли, в которую она включена, а также права, данные группе PUBLIC. Поэтому, например, лишение PUBLIC права SELECT не обязательно будет означать, что все роли лишатся права SELECT для данного объекта. Оно сохранится у тех ролей, которым дано непосредственно или косвенно, через другую роль. Подобным образом, лишение права SELECT какого-либо пользователя может не повлиять на его возможность пользоваться правом SELECT, если это право дано группе PUBLIC или другой роли, в которую он включён.

Если указано GRANT OPTION FOR, отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.

Если пользователь обладает правом с правом передачи и он дал его другим пользователям, оно считается зависимым (inherited). Когда первый пользователь лишается самого права или права передачи и существуют зависимые права, эти зависимые права также отзываются, если дополнительно указано CASCADE; в противном случае операция завершается ошибкой.

Когда отзывается право доступа к таблице, с ним вместе автоматически отзываются соответствующие права для каждого столбца таблицы (если такие права заданы). С другой стороны, если роли были даны права для таблицы, лишение роли таких же прав на уровне отдельных столбцов ни на что не влияет.

---

[117] REVOKE URL: https://www.postgresql.org/docs/18/sql-revoke.html (дата обращения 19.03.2026) [7]

Также обратите внимание, что пользователь может отзывать только те права, которые он дал другому непосредственно. Если, например, пользователь A дал право с правом передачи пользователю B, а пользователь B, в свою очередь, дал это право пользователю C, то пользователь A не сможет лишить этого права непосредственно C.

Вместо этого пользователь A может лишить права передачи прав пользователя B и использовать параметр **CASCADE**, чтобы этого права по цепочке лишился пользователь C. Или же, например, если и A, и B дали одно и то же право C, то A сможет отозвать право, которое дал он, но не пользователь B, так что в результате C всё равно будет иметь это право.

Рассмотрим на практике ряд тонкостей.

Создадим новую базу данных **testdb** и зайдём в созданную базу данных под пользователем **postgres**:

```
CREATE DATABASE testdb;
\c testdb
```

Создадим новую схему testnm:

```
CREATE SCHEMA testnm;
```

Создадим новую таблицу **t1** с одной колонкой c1 типа integer:

```
CREATE TABLE t1(c1 integer);
```

Вставим строку со значением 1:

```
INSERT INTO t1 VALUES (1);
```

```
postgres=# CREATE DATABASE testdb;
CREATE DATABASE
postgres=# \c testdb
You are now connected to database "testdb" as user "postgres".
testdb=# CREATE SCHEMA testnm;
CREATE SCHEMA
testdb=# CREATE TABLE t1(c1 integer);
CREATE TABLE
testdb=# INSERT INTO t1 VALUES (1);
INSERT 0 1
```

Создадим новую роль readonly:

```
CREATE ROLE readonly;
```

Дадим новой роли право на подключение к базе данных **testdb**:
```
GRANT CONNECT ON DATABASE testdb TO readonly;
```

Дадим новой роли право на использование схемы **testnm**:
```
GRANT USAGE ON SCHEMA testnm TO readonly;
```

И право на select для всех таблиц схемы **testnm**:
```
GRANT SELECT ON ALL TABLES IN SCHEMA testnm TO readonly;
```

```
testdb=# CREATE ROLE readonly;
CREATE ROLE
testdb=# GRANT CONNECT ON DATABASE testdb TO readonly;
GRANT
testdb=# GRANT USAGE ON SCHEMA testnm TO readonly;
GRANT
testdb=# GRANT SELECT ON ALL TABLES IN SCHEMA testnm TO readonly;
GRANT
```

Создадим пользователя **testread** с паролем test123:
```
CREATE USER testread WITH PASSWORD 'test123';
```

Предоставим роль readonly пользователю **testread**:
```
GRANT readonly TO testread;
```

Зайдём под пользователем **testread** в базу данных **testdb**:
```
\c testdb testread
```

Выберем все записи из таблицы **t1**:
```
SELECT * FROM t1;
```

```
testdb=# CREATE USER testread WITH PASSWORD 'test123';
CREATE ROLE
testdb=# GRANT readonly TO testread;
GRANT ROLE
testdb=# \c testdb testread
Password for user testread:
You are now connected to database "testdb" as user "testread".
testdb=> SELECT * FROM t1;
ERROR:  permission denied for table t1
```

Почему нет доступа?

Посмотрим на список таблиц:
```
\dt
```

```
testdb=> \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | t1   | table | postgres
(1 row)
```

Всё просто. Вспоминаем про **search_path** – таблица создана в схеме **public**, а не **testnm**. Прав на public для роли **readonly** не давали.

Вернёмся в базу данных **testdb** под пользователем **postgres**:

```
\c testdb postgres
```

Удалим таблицу t1:

```
DROP TABLE t1;
```

Создадим её заново, но уже с явным указанием имени схемы **testnm**:

```
CREATE TABLE testnm.t1(c1 integer);
```

Вставим строку со значением 1 в таблицу **t1** с указанием схемы **testnm**:

```
INSERT INTO testnm.t1 VALUES (1);
```

```
testdb=# \c testdb postgres
You are now connected to database "testdb" as user "postgres".
testdb=# DROP TABLE t1;
DROP TABLE
testdb=# CREATE TABLE testnm.t1(c1 integer);
CREATE TABLE
testdb=# INSERT INTO testnm.t1 VALUES (1);
INSERT 0 1
```

Зайдём под пользователем **testread** в базу данных **testdb**:

```
\c testdb testread
```

Выберем все записи из таблицы **t1**:

```
SELECT * FROM testnm.t1;
```

```
testdb=# \c testdb testread
Password for user testread:
You are now connected to database "testdb" as user "testread".
testdb=> SELECT * FROM testnm.t1;
ERROR:  permission denied for table t1
```

И опять нет доступа. Как думаете, почему?

Потому что `GRANT SELECT ON ALL TABLES IN SCHEMA testnm TO readonly;` дал доступ только для существующих на тот момент времени таблиц, а таблица **t1** пересоздавалась.

Что же сделать, чтобы такое больше не повторялось? Зададим права по умолчанию:

```
\c testdb postgres;
ALTER DEFAULT PRIVILEGES IN SCHEMA testnm GRANT SELECT ON TABLES TO readonly;
\c testdb testread;
SELECT * FROM testnm.t1;
```

```
testdb=> \c testdb postgres;
You are now connected to database "testdb" as user "postgres".
testdb=# ALTER DEFAULT PRIVILEGES IN SCHEMA testnm GRANT SELECT ON TABLES TO readonly;
ALTER DEFAULT PRIVILEGES
testdb=# \c testdb testread;
Password for user testread:
You are now connected to database "testdb" as user "testread".
testdb=> SELECT * FROM testnm.t1;
ERROR:  permission denied for table t1
```

И опять нет доступа…

Потому что **ALTER DEFAULT будет действовать только для новых таблиц**. Надо пересоздать таблицу или grant select на существующую.

Рассмотрим другой аспект – про безопасность.
До PostgreSQL 15 мог бы быть данный сценарий:
Создадим другую таблицу под пользователем testdb:

```
CREATE TABLE t2(c1 integer);
```

Добавим туда одну запись:

```
INSERT INTO t2 VALUES (2);
```

```
testdb=> CREATE TABLE t2(c1 integer);
CREATE TABLE
testdb=> INSERT INTO t2 VALUES (2);
INSERT 0 1
```

Всё получилось, **НО** нам же никто прав не давал на создание таблиц и insert в них под ролью readonly?

Это всё потому, что **search_path** указывает в первую очередь на схему public. А схема public создаётся в каждой базе данных по умолчанию. И grant на все действия в этой схеме даётся роли public. А роль public добавляется всем новым пользователям. Соответственно, каждый пользователь может по умолчанию создавать объекты в схеме public любой базы данных, естественно, если у него есть право на подключение к этой базе данных.

Чтобы раз и навсегда забыть про роль public – а в продакшн базе данных про неё лучше забыть – выполните следующие действия:

```
\c testdb postgres;
REVOKE CREATE ON SCHEMA public FROM public;
REVOKE ALL ON DATABASE testdb FROM public;
\c testdb testread;
```

```
testdb=> \c testdb postgres;
You are now connected to database "testdb" as user "postgres".
testdb=# REVOKE CREATE ON SCHEMA public FROM public;
REVOKE
testdb=# REVOKE ALL ON DATABASE testdb FROM public;
REVOKE
testdb=# \c testdb testread;
Password for user testread:
You are now connected to database "testdb" as user "testread".
```

И попробуем создать таблицу:

```
CREATE TABLE t3(c1 integer);
```

```
testdb=> CREATE TABLE t3(c1 integer);
ERROR:  permission denied for schema public
LINE 1: CREATE TABLE t3(c1 integer);
                      ^
```

Ожидаемо, ничего не получилось.

С версии 15 отозвали права по умолчанию на public для новых пользователей без явного на то указания, что очень правильно!

Временную таблицу тоже создать не получится:

```
CREATE TEMP TABLE t3(c1 integer);
```

```
testdb=> CREATE TEMP TABLE t3(c1 integer);
ERROR:  permission denied to create temporary tables in database "testdb"
LINE 1: CREATE TEMP TABLE t3(c1 integer);
                          ^
```

После того, как обсудили базовое управление пользователями, необходимо затронуть стратегии выдачи прав.

Есть три основных модели (и много комбинаций):

1. RBAC внутри PostgreSQL (можно прикрутить LDAP[118], AD[119], IPA[120] и другие):

- можем дотюнить доступ вплоть до колонки
- несмотря на сложность настройки, в целом, при интеграции с LDAP есть и плюсы, и тяжесть сопоставления ролей
- излишне ветвистая структура прав доступа – легко что-нибудь упустить из вида
- сложность изменения структуры БД – необходимо помнить про групповые права на LDAP
- потеря производительности – проверка доступа каждый раз при обращении к тому или иному объекту
  - для интеграции с ldap рекомендую ldap2pg[121]
  - для бесшовной интеграции OAUTH2[122] подойдёт keycloak[123]

2. Одна или несколько групповых учётных записей и RBAC на бэкенде (перекладываем разграничение прав доступа на приложение):

- очевидная лёгкость настройки и простота
- функционал пользователя ограничиваем на бэке
- дополнительный слой логики при проектировании приложения
- недоступные функции просто у пользователя не будут отображаться
- не забываем заблокировать учётную запись уволенного сотрудника
- предпочтительный вариант
- бэкендеры будут против, т.к. повышает сложность кодовой базы

3. Набор хранимых процедур с нужными правами (security definer) – напрямую к таблицам никто доступа не имеет:

- можем менять структуру независимо от приложения – главное имя хранимой процедуры сохранить, а что внутри – пользователя не касается
- легко вводить новый функционал или убирать старый

---

[118] LDAP URL: https://www.onelogin.com/learn/what-is-ldap (дата обращения 20.03.2026) [8]
[119] AD URL: https://en.wikipedia.org/wiki/Active_Directory (дата обращения 20.03.2026) [9]
[120] AD vs IPA URL: https://habr.com/ru/companies/astralinux/articles/770418/ (дата обращения 20.03.2026) [10]
[121] ldap2pg URL: https://pgconf.ru/talk/2451865 (дата обращения 16.03.2026) [11]
[122] oauth2 URL: https://oauth.net/2/ (дата обращения 16.03.2026) [12]
[123] keycloak URL: https://habr.com/ru/companies/tantor/articles/923582/ (дата обращения 16.03.2026) [13]

- ❖ дополнительный слой абстракции, платим производительностью
- ❖ можем получить сайд-эффекты после изменений цепочки вызовов, особенно если этим занимаются разные команды

Стоит отметить, что кроме устоявшейся модели RBAC, существуют и альтернативы: ABAC и PERM[124].

Скорее всего модель ограничения прав будет выбрана в соответствии с требованиями СБ, но если есть выбор, то второй вариант доставит минимум нагрузки.

Тут всё упирается в ресурсы: людей для поддержки железа, ПО и финансовые возможности компании.

Когда всё на одном сервере и на одной БД – удобно, но если что-то падает – это глобальный простой.

Опять же, а если надо откатить данные только одной схемы?

Глобально зависит от заложенной архитектуры, RTO, RPO[125], SLA[126], квалификации персонала и множества других факторов.

RLS (row level security).

Важным моментом в настройке прав доступа является возможность защитить данные вплоть до строк. Например, проверять владельца записи и не предоставлять доступ другим группам. Для этого используется механизм RLS (row level security).

Для начала данный механизм должен быть включён[127] на конкретной таблице, потому что по умолчанию он не используется. Далее необходимо создать разрешительную или запретительную политику, где прописать правила доступа.

Посмотрим на практике.

Создание объектов – таблица depart со списком пользователей и их отделов и таблица “Доход” с отделами и доходами:

```
DROP TABLE IF EXISTS depart;
CREATE TABLE depart(login text, department text);
```

[124] PERM URL: https://habr.com/ru/articles/539778/ (дата обращения 20.03.2026) [14]
[125] TRO vs RPO URL: https://www.druva.com/blog/understanding-rpo-and-rto (дата обращения 20.03.2026) [15]
[126] SLA URL: https://www.techtarget.com/searchitchannel/definition/service-level-agreement (дата обращения 20.03.2026) [16]
[127] row security URL: https://www.postgresql.org/docs/current/ddl-rowsecurity.html (дата обращения 16.03.2026) [17]

```
INSERT INTO depart VALUES ('eugene', 'CEO'), ('alex', 'Sales'),('ivan', 'Sales');

DROP TABLE IF EXISTS revenue;
CREATE TABLE revenue(department text, amount numeric(10,2));
INSERT INTO revenue(department,amount) SELECT 'CEO', random()* 10000.00 FROM generate_series(1,10);
INSERT INTO revenue(department,amount) SELECT 'Sales', random()* 100.00 FROM generate_series(1,1000);
```

Создадим политику проверки – название отдела (department из таблицы revenue) должно совпадать с отделом (department) в соответствии с именем пользователя из таблички depart:

```
CREATE POLICY departments ON revenue USING (department = (SELECT department FROM depart WHERE login = current_user));
```

Включим RLS на таблице revenue:

```
ALTER TABLE revenue ENABLE ROW LEVEL SECURITY;
```

Создадим пользователей для проверки работоспособности:

```
DROP USER IF EXISTS eugene;
CREATE USER eugene WITH PASSWORD 'test123';
GRANT SELECT ON depart, revenue TO eugene;
DROP USER IF EXISTS alex;
CREATE USER alex WITH PASSWORD 'test123';
GRANT SELECT ON depart, revenue TO alex;
```

Проверим под правами суперпользователя (для него не проверяются никакие ограничения, в том числе RLS, кроме прав на хранимые функции и процедуры с правами вызывателя):

```
SELECT department, SUM(amount) FROM revenue GROUP BY department;
```

```
postgres=# SELECT department, SUM(amount) FROM revenue GROUP BY department;
 department |   sum
------------+----------
 CEO        | 53732.42
 Sales      | 50481.46
(2 rows)
```

Проверим под правами пользователя alex:

```
\c - alex localhost 5432
SELECT department, SUM(amount) FROM revenue GROUP BY department;
```

```
You are now connected to database "postgres" as user "alex" on host "localhost"
postgres=> SELECT department, SUM(amount) FROM revenue GROUP BY department;
 department |   sum
------------+----------
 Sales      | 50481.46
(1 row)
```

Проверим под правами пользователя eugene:

```
\c - eugene localhost 5432
SELECT department, SUM(amount) FROM revenue GROUP BY department;
```

```
You are now connected to database "postgres" as user "eugene".
postgres=> SELECT department, SUM(amount) FROM revenue GROUP BY department;
 department |   sum
------------+----------
 CEO        | 53732.42
(1 row)
```

Пользователи видят только те строки, к которым имеют доступ.

Создание запретительной политики для новых объектов (под пользователем postgres). Для этого создадим новую ограничительную политику (AS RESTRICTIVE):

```
CREATE POLICY amount ON revenue AS RESTRICTIVE USING (true) WITH CHECK (abs(amount) <= 1000.00);
```

В команде выше два условия:

- **USING** (true) — все существующие строки видны в любом случае
- **WITH CHECK** (abs(amount) <= 100.00) — новые строки должны быть не более 1000

И дадим alex ещё привилегию на вставку в эту таблицу (переключившись для этого на пользователя postgres):

```
GRANT INSERT ON revenue TO alex;
```

Проверим, какие данные получится вставить:

```
\c - alex localhost 5432
INSERT INTO revenue VALUES ('Sales', 100);
```

```
INSERT INTO revenue VALUES ('CEO', 100);
INSERT INTO revenue VALUES ('test', 100);
INSERT INTO revenue VALUES ('Sales', 1001);
```

```
You are now connected to database "postgres" as user "alex" on host "localhost"
postgres=> INSERT INTO revenue VALUES ('Sales', 100);
INSERT 0 1
postgres=> INSERT INTO revenue VALUES ('CEO', 100);
ERROR:  new row violates row-level security policy for table "revenue"
postgres=> INSERT INTO revenue VALUES ('test', 100);
ERROR:  new row violates row-level security policy for table "revenue"
postgres=> INSERT INTO revenue VALUES ('Sales', 1001);
ERROR:  new row violates row-level security policy "amount" for table "revenue"
```

Попробуем то же самое для пользователя eugene (права выдадим под суперпользователем):

```
GRANT INSERT ON revenue TO eugene;
\c - eugene localhost 5432
INSERT INTO revenue VALUES ('CEO', 10000);
INSERT INTO revenue VALUES ('CEO', 100);
```

```
You are now connected to database "postgres" as user "eugene" on host "localhost"
postgres=> INSERT INTO revenue VALUES ('CEO', 10000);
ERROR:  new row violates row-level security policy "amount" for table "revenue"
postgres=> INSERT INTO revenue VALUES ('CEO', 100);
INSERT 0 1
```

Всё работает, как и ожидали.

*Важно! Необходимо очень аккуратно использовать представления (**view**) при использовании RLS, так как эта возможность имеет известную уязвимость!*

Посмотрим на практике, создав представления:

```
CREATE OR REPLACE VIEW tv1 as SELECT department, SUM(amount) FROM revenue GROUP BY department;
GRANT SELECT ON tv1 TO eugene;
\c - eugene localhost 5432
SELECT * FROM tv1;
```

```
postgres=# CREATE OR REPLACE VIEW tv1 as SELECT department, SUM(am
CREATE VIEW
postgres=# GRANT SELECT ON tv1 TO eugene;
GRANT
postgres=# \c - eugene localhost 5432
Password for user eugene:
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384,
You are now connected to database "postgres" as user "eugene" on h
postgres=> SELECT * FROM tv1;
 department |   sum
------------+----------
 CEO        | 53832.42
 Sales      | 50581.46
(2 rows)
```

И получили полный доступ – RLS не отработал!

Чтобы избежать такой проблемы, существует опция **security_barrier**:

```
CREATE OR REPLACE VIEW tv2 WITH (security_barrier) as SELECT department, SUM(amount) FROM revenue GROUP BY department;
GRANT SELECT ON tv2 TO eugene;
\c - eugene localhost 5432
SELECT * FROM tv2;
```

```
postgres=# CREATE OR REPLACE VIEW tv2 WITH (security_barrier)
GRANT SELECT ON tv2 TO eugene;
\c - eugene localhost 5432
CREATE VIEW
GRANT
Password for user eugene:
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SH
You are now connected to database "postgres" as user "eugene"
postgres=> SELECT * FROM tv2;
 department |   sum
------------+----------
 CEO        | 53832.42
 Sales      | 50581.46
(2 rows)
```

Но она не помогает! Если внимательно посмотреть документацию, то необходимо ещё включить каскадную проверку RLS **WITH CASCADED CHECK OPTION**:

```
CREATE OR REPLACE VIEW tv4 WITH (security_barrier) as SELECT * FROM revenue WITH CASCADED CHECK OPTION;
GRANT SELECT ON tv4 TO eugene;
\c - eugene localhost 5432
SELECT * FROM tv4;
```

```
 CEO        | 2008.40
 CEO        | 1536.01
 CEO        | 4391.21
 Sales      |   19.05
 Sales      |   52.83
 Sales      |   77.71
 Sales      |   85.36
```

Как ни странно, но документация не соответствует и нужно указывать security_invoker:

```
CREATE OR REPLACE VIEW tv5 WITH (security_invoker) as SELECT * FROM revenue WITH CASCADED CHECK OPTION;
GRANT SELECT ON tv5 TO eugene;
\c - eugene localhost 5432
SELECT * FROM tv5;
```

```
postgres=> SELECT * FROM tv5;
 department | amount
------------+---------
 CEO        | 9300.91
 CEO        | 2214.13
 CEO        | 9288.13
 CEO        | 2838.86
 CEO        | 6263.50
 CEO        | 7867.19
 CEO        | 8024.08
 CEO        | 2008.40
 CEO        | 1536.01
 CEO        | 4391.21
 CEO        |  100.00
(11 rows)
```

Так что нужно подходить очень осознанно, когда используете RLS!
Естественно, **view as select * from view;** имеет те же проблемы.

! Помечено, как LEAKPROOF[128]!

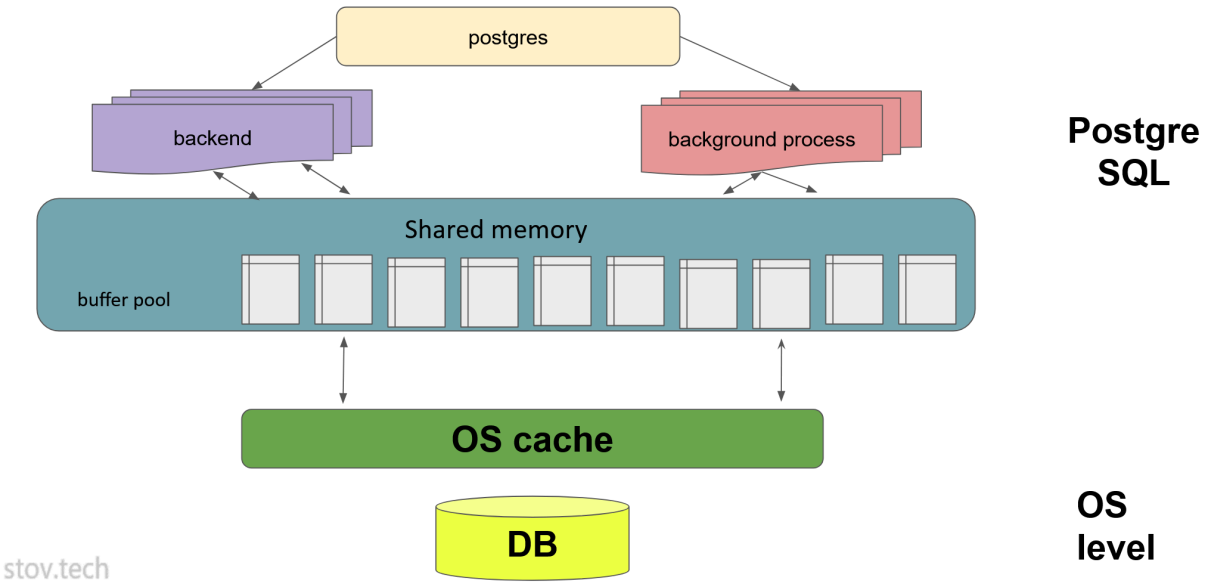
В этой главе обсудили, как выдать и забрать права на доступ к объектам. Как изменить права по умолчанию из соображений безопасности. Варианты выбора стратегий управления пользователями. В следующей главе будем рассматривать, что такое журналы и для чего они нужны.

Все команды, рассмотренные в этой главе, вы можете найти на гитхабе в файле https://github.com/aeuge/postgres18book/blob/main/scripts/07/users.sql

[128] привелегии URL: https://www.postgresql.org/docs/current/rules-privileges.html (дата обращения 16.03.2026) [18]

8. Shared buffers, WAL, bgwriter, checkpoint

В этой главе будем рассматривать журналы и зачем они нужны. Рассмотрим графически, как в PostgreSQL принципиально выглядит схема работы с данными:



Обмен всеми данными между процессами идёт через буферный кэш, находящийся в оперативной памяти сервера. Если данных в кэше не хватает для обработки запросов, то они считываются с диска. Все операции DML изменяют эти буферы и выгружаются на диск по определённому алгоритму. Зачем их выгружать?

Попробуйте ответить на этот вопрос самостоятельно, дальше вы получите на него ответ.

В этой схеме есть два момента:

- оперативная память очень быстра, но её мало
- жёсткий диск (RAID[129], SSD[130], NVMe[131]) огромный, но медленный

Давайте разберём логику работы этой схемы подробнее.

[129] RAID URL: https://en.wikipedia.org/wiki/RAID (дата обращения 21.03.2026) [1]
[130] SSD URL: https://en.wikipedia.org/wiki/Solid-state_drive (дата обращения 21.03.2026) [2]
[131] NVMe URL: https://en.wikipedia.org/wiki/NVM_Express (дата обращения 21.03.2026) [3]