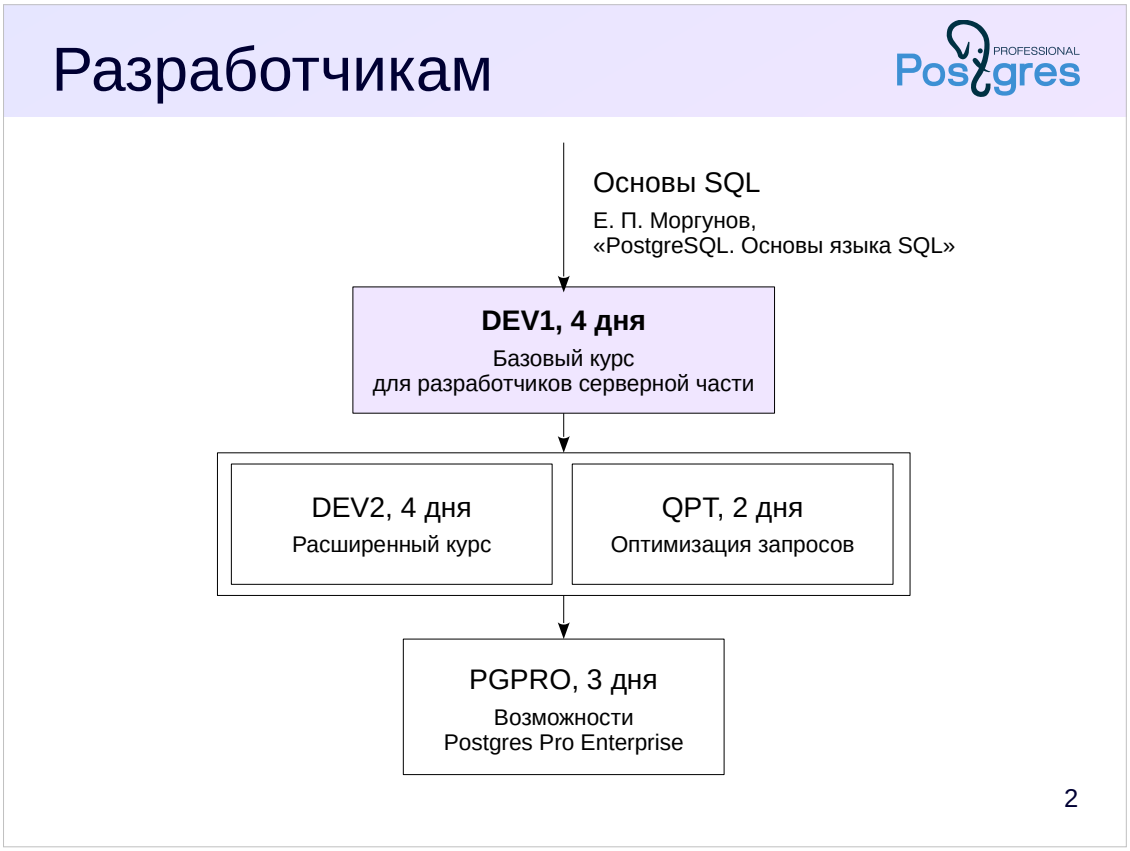
Разработка серверной части приложений

Базовый курс



Для разработчиков серверной части приложений мы предлагаем специальную линейку курсов.

Для прохождения этих курсов необходимы предварительные знания основ языка **SQL**. Специального курса по языку SQL в нашей линейке нет, но существует множество книг и других образовательных ресурсов, с помощью которых можно освоить SQL. Мы рекомендуем книгу Евгения Моргунова «PostgreSQL. Основы языка SQL»:

https://postgrespro.ru/education/books/sqlprimer

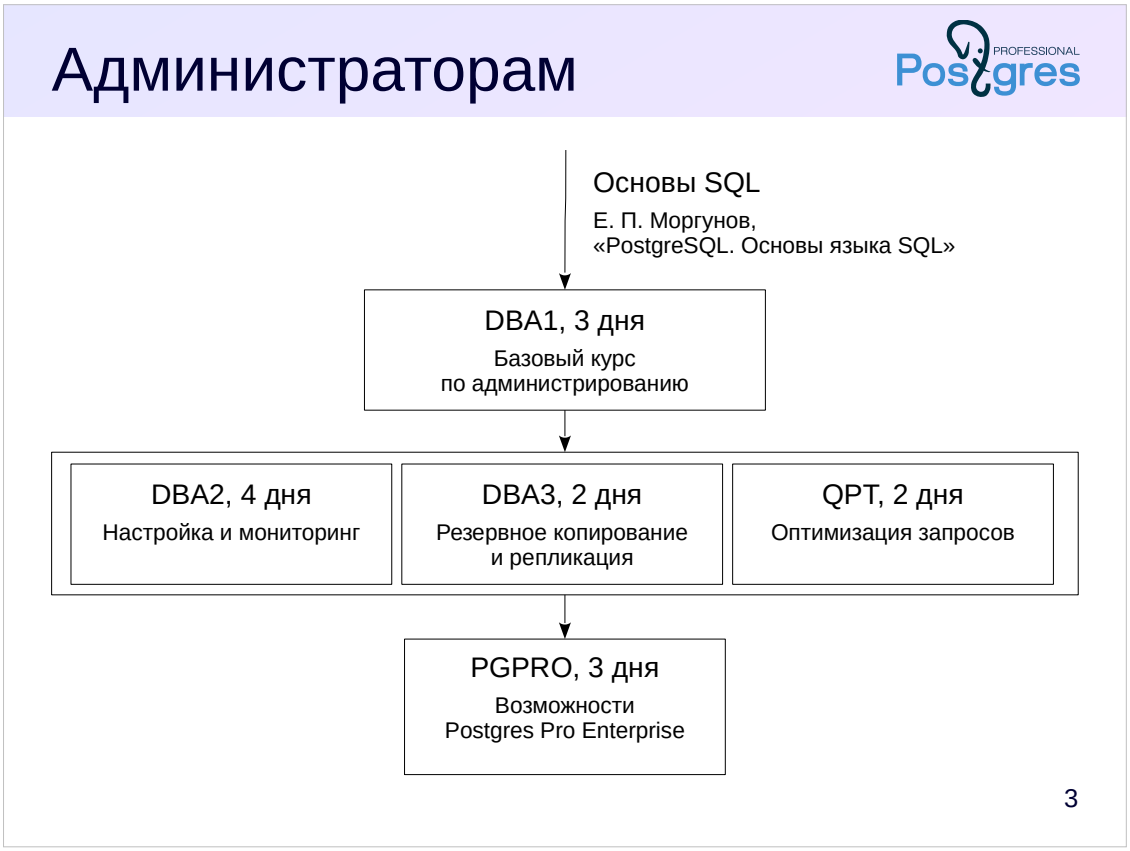
Базовым курсом для разработчиков является данный курс **DEV1**.

В курсе **DEV2** рассматриваются особенности внутреннего устройства сервера, влияющие на разработку прикладного кода, и всесторонне изучается расширяемость: возможность дополнить серверные механизмы собственным кодом, что позволяет использовать PostgreSQL для решения самых разнообразных задач.

В курсе **QPT** детально разбираются механизмы планирования и выполнения запросов, рассматривается настройка параметров экземпляра, связанных с производительностью, и изучаются возможности для поиска проблемных запросов и их оптимизации.

В курсе **PGPRO** рассматриваются дополнительные возможности, которые предоставляет СУБД Postgres Pro Enterprise.

https://postgrespro.ru/education/courses



Для администраторов мы предлагаем следующие курсы.

В базовом курсе **DBA1** даются общие сведения об архитектуре PostgreSQL, процессе установки, базовой настройке и управлении сервером. Рассматриваются основные задачи администрирования, вопросы управления доступом. Приводится обзор резервного копирования и репликации.

В курсе **DBA2** обсуждается настройка различных конфигурационных параметров исходя из понимания внутренней организации сервера; говорится о мониторинге сервера с использованием обратной связи для итеративной настройки параметров. Также рассматриваются настройки, связанные с локализацией, управление расширениями и знакомство с процедурой обновления сервера.

Курс **DBA3** посвящен рассмотрению резервного копирования, а также настройкам физической и логической репликации и сценариям ее использования. Также дается общее представление о способах и сложностях построения высокодоступных, масштабируемых кластеров.

Курсы **QPT** и **PGPRO** являются общими для разработчиков и администраторов.

Курсы по администрированию могут быть интересны и разработчикам, желающим детальнее изучить внутреннее устройство PostgreSQL, а также в случае, когда на проекте нет выделенной роли администратора.

О курсе

Продолжительность: 3 дня

Предварительные знания

- основы SQL
- опыт работы с любым процедурным языком программирования
- минимальные сведения о работе в Unix

Какие навыки будут получены

- общие сведения об архитектуре PostgreSQL
- использование основных объектов БД: таблиц, индексов, представлений
- программирование на стороне сервера на языках SQL и PL/pgSQL
- использование основных типов данных, включая записи и массивы
- организация взаимодействия с клиентской частью приложения

Базовый курс знакомит разработчиков приложений, работающих над серверной частью, с основами PostgreSQL и написанием хранимых процедур и функций на языках SQL и PL/pgSQL.

Используемые материалы

Подготовленная виртуальная машина

- ОС Xubuntu 22
- PostgreSQL 16 с документацией на русском языке
- учебное веб-приложение «Книжный магазин»
- pgAdmin 4

Учебные материалы

- руководство слушателя
- презентации, демонстрации, практические задания и их решение (в форматах html и pdf)
- справочные материалы — функции и типы данных PostgreSQL, схема основных таблиц системного каталога с командами psql, некоторые команды Unix, настройка pgAdmin

Если вы проходите курс самостоятельно, обязательно начните со знакомства с Руководством слушателя. В числе прочего в нем написано, где скачать и как использовать виртуальную машину курса и прочие материалы. Все материалы курса доступны по адресу:

https://postgrespro.ru/education/courses/DEV1

Выполнение практических заданий очень важно для получения навыков работы с PostgreSQL. Обязательно старайтесь сначала самостоятельно выполнить задания, а затем просмотрите предлагаемые нами решения, даже если задание не вызвало вопросов. В решениях могут содержаться дополнительные сведения, которые не упоминаются в презентациях и демонстрациях.

Учебные материалы (презентации, демонстрации, практические задания и их решения) доступны в двух форматах. Формат html удобен для онлайн-работы, он позволяет копировать фрагменты текста и кода. Документ в формате pdf разбит на страницы и удобен для печати.

Дополнительные справочные материалы помогут быстро найти нужную информацию.

В виртуальной машине установлен pgAdmin 4. В курсе мы используем только psql, но при желании можно воспользоваться и графической средой.

Организационные вопросы

День: ~8 академических часов + обед (1час)

Каждая тема, как правило, состоит из

- презентации и демонстраций: ~25–60 мин
- практических заданий: ~20–30 мин, включая перерыв

День 1

Postgres PROFESSIONAL

Базовый инструментарий

01. Установка и управление, psql

Архитектура

02. Общее устройство PostgreSQL

Организация данных

03. Логическая структура

04. Физическая структура

Первый день занятий посвящен в основном теоретической подготовке. Здесь рассматриваются основы архитектуры PostgreSQL, без знания которых невозможно полноценное использование СУБД. Полученные знания будут применяться на практике в последующих темах курса.

День 2

Приложение «Книжный магазин» 

07. Схема данных и интерфейс

SQL

08. Функции
09. Процедуры
10. Составные типы

PL/pgSQL

11. Обзор и конструкции языка

Начиная с модуля «Приложение «Книжный магазин»» второго дня занятий каждая тема содержит два набора практических заданий: основных, связанных с этим приложением (они отмечены значком книги) и дополнительных. За время, которое отводится на задания (около 30 минут), выполнить всю практику невозможно. Используйте дополнительные задания для самостоятельной работы.

День 3

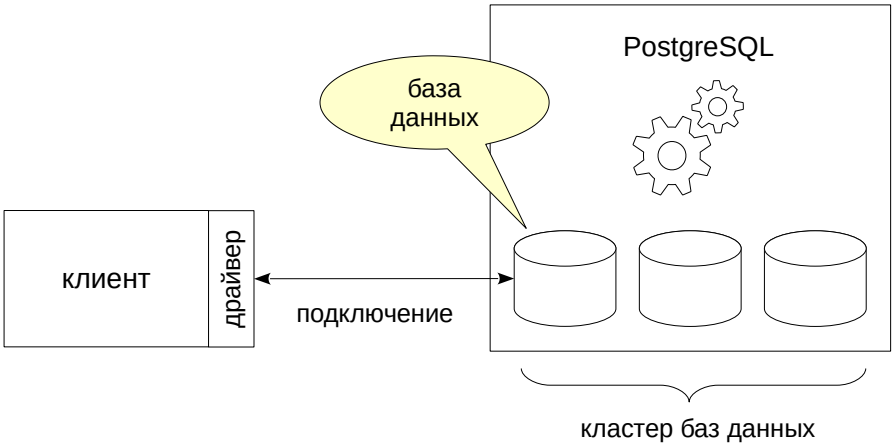
PL/pgSQL (продолжение)

12. Выполнение запросов
13. Курсоры
14. Динамические команды
15. Массивы
16. Обработка ошибок
17. Триггеры
18. Отладка

Обзор базового инструментария

Установка и управление, psql

Кластер БД, сервер, клиент



Начнем с основных понятий.

PostgreSQL — программа, которая относится к классу *систем управления базами данных*.

Когда эта программа выполняется, мы называем ее *сервером* PostgreSQL или *экземпляром сервера*. Пока сервер представляется для нас «черным ящиком», но постепенно мы познакомимся с тем, как он устроен.

Данные, которыми управляет PostgreSQL, хранятся в *базах данных*. Один экземпляр PostgreSQL одновременно работает с несколькими базами данных. Этот набор баз данных называется *кластером баз данных*. Подробнее мы будем говорить о базах данных в теме «Организация данных. Логическая структура».

С сервером взаимодействуют клиенты — внешние приложения, которые могут подключаться к одной из баз сервера и посылать *запросы* для выполнения.

Итак: кластер баз данных — это данные в файлах; сервер или экземпляр сервера — программа, управляющая кластером баз данных, а клиент — программа, позволяющая «общаться» с сервером.

Установка

Postgres PROFESSIONAL

Варианты

- готовые пакеты (предпочтительный способ)
- установка из исходных кодов
- без установки — облачные сервисы

Расширения

- дополнительный функционал
- устанавливаются отдельно
- в поставке с сервером — модули и программы (~50 штук)

Предпочтительный вариант установки PostgreSQL — использование пакетных менеджеров (таких, как apt или rpm) и готовых пакетов. В этом случае получается понятная, поддерживаемая и легко обновляемая установка. Пакеты существуют для большинства операционных систем.

Другой вариант — самостоятельная сборка PostgreSQL из исходных кодов. Это может понадобиться для установки нестандартных значений параметров или при использовании не популярной платформы.

Готовые пакеты и исходные коды: http://www.postgresql.org/download/

Кроме того, можно использовать готовые облачные решения, что позволяет обойтись вообще без установки. Такую возможность дают многие ведущие зарубежные (Amazon RDS, Google Cloud SQL, Microsoft Azure) и отечественные (Yandex Cloud, Облако Mail.ru) платформы.

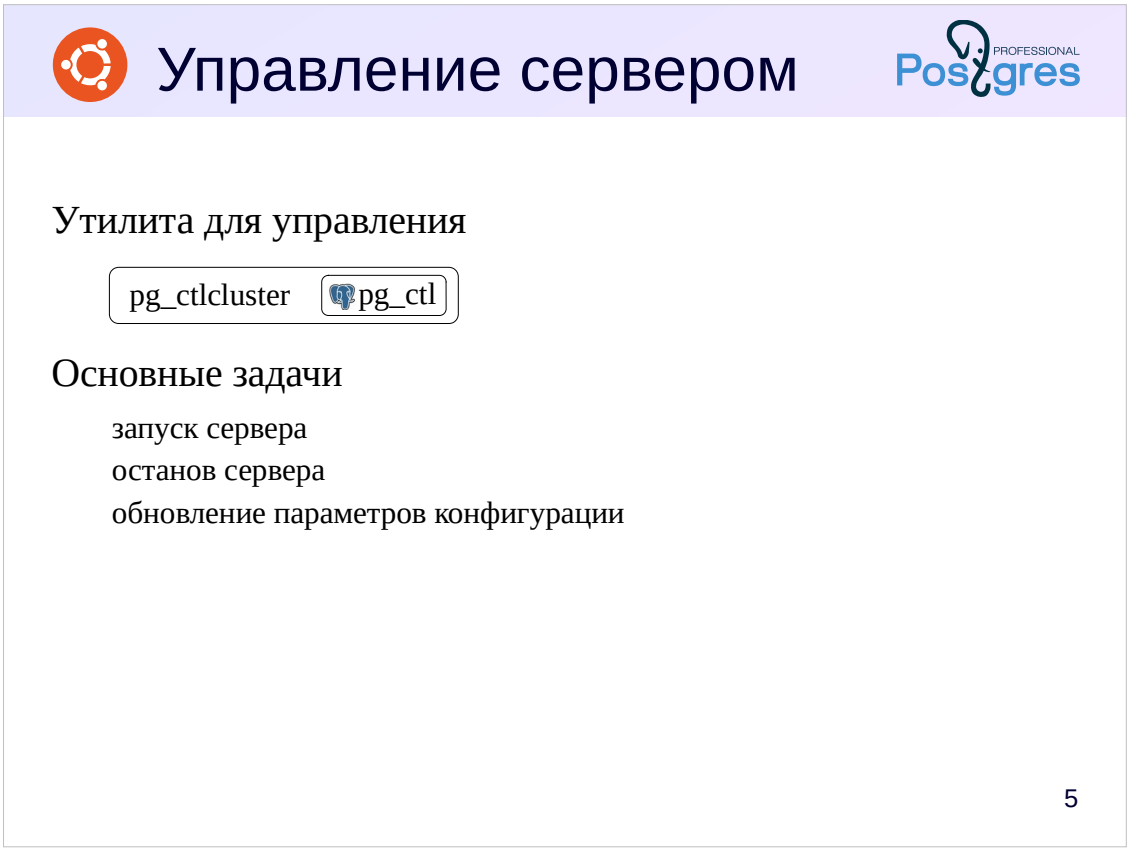
В курсе мы будем использовать виртуальную машину с ОС Xubuntu 22 и сервер PostgreSQL 16, установленный из пакета для этой ОС. В этом случае сразу настраивается автоматический запуск и останов сервера PostgreSQL при запуске и останове операционной системы.

Для PostgreSQL существует большое количество расширений, которые подключают новый функционал к СУБД «на лету», без изменения ядра системы. В состав дистрибутива входит 50 расширений.

https://postgrespro.ru/docs/postgresql/16/contrib

https://postgrespro.ru/docs/postgresql/16/contrib-prog

Список доступных расширений и статус их установки можно посмотреть в представлении pg_available_extensions.



К основным операциям управления сервером относятся начальная инициализация кластера баз данных, запуск и останов сервера, обновление конфигурации и некоторые другие. Для выполнения этих действий предназначена утилита pg_ctl, входящая в состав PostgreSQL.

В пакетном дистрибутиве для Ubuntu доступ к утилите pg_ctl осуществляется не напрямую, а через специальную обертку pg_ctlcluster. Справку по использованию pg_ctlcluster можно получить командой:

$ man pg_ctlcluster

Также можно получить информацию об установленных кластерах и их текущем состоянии при помощи команд:

$ pg_lsclusters

$ pg_ctlcluster status

Более подробная информация об управлении сервером для администраторов баз данных:

https://postgrespro.ru/docs/postgresql/16/app-pg-ctl

https://postgrespro.ru/docs/postgresql/16/runtime

Установка и управление

В виртуальной машине курса установка выполнена из пакета. Каталог установки PostgreSQL:

```
student$ ls -l /usr/lib/postgresql/16
```

```
total 8
drwxr-xr-x 2 root root 4096 июн 21 16:28 bin
drwxr-xr-x 4 root root 4096 июл  3 15:56 lib
```

Владелец ПО сервера — пользователь root.

Кластер баз данных автоматически инициализируется при установке из пакета и находится в каталоге /var/lib/postgresql/16/main.

В последующих темах мы будем ссылаться на этот каталог как PGDATA, по имени переменной ОС, которую можно установить для использования в некоторых утилитах сервера.

Владельцем каталога является пользователь postgres. Вот содержимое каталога:

```
student$ sudo ls -l /var/lib/postgresql/16/main
```

```
total 96
drwx------ 1 postgres postgres 4096 июл  5 16:17 base
drwx------ 1 postgres postgres 4096 июл  5 16:17 global
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_commit_ts
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_dynshmem
drwx------ 4 postgres postgres 4096 июл  3 15:57 pg_logical
drwx------ 4 postgres postgres 4096 июл  3 15:57 pg_multixact
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_notify
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_replslot
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_serial
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_snapshots
drwx------ 1 postgres postgres 4096 июл  5 16:17 pg_stat
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_stat_tmp
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_subtrans
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_tblspc
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_twophase
-rw------- 1 postgres postgres    3 июл  3 15:57 PG_VERSION
drwx------ 1 postgres postgres 4096 июл  3 15:57 pg_wal
drwx------ 2 postgres postgres 4096 июл  3 15:57 pg_xact
-rw------- 1 postgres postgres   88 июл  3 15:57 postgresql.auto.conf
-rw------- 1 postgres postgres  130 июл  5 16:17 postmaster.opts
-rw------- 1 postgres postgres  108 июл  5 16:17 postmaster.pid
```

При установке из пакета в настройки запуска ОС добавляется автоматический запуск PostgreSQL. Поэтому после загрузки операционной системы отдельно стартовать PostgreSQL не нужно.

Можно явным образом управлять сервером с помощью следующих команд, которые выдаются от имени привилегированного пользователя ОС через sudo:

Остановить сервер:

```
student$ sudo pg_ctlcluster 16 main stop
```

Запустить сервер:

```
student$ sudo pg_ctlcluster 16 main start
```

Перезапустить:

```
student$ sudo pg_ctlcluster 16 main restart
```

Обновить конфигурацию:

```
student$ sudo pg_ctlcluster 16 main reload
```

Получить информацию о сервере:

```
student$ sudo pg_ctlcluster 16 main status
```

```
pg_ctl: server is running (PID: 3440)
/usr/lib/postgresql/16/bin/postgres "-D" "/var/lib/postgresql/16/main" "-c"
"config_file=/etc/postgresql/16/main/postgresql.conf"
```

Список установленных экземпляров (можно без sudo):

```
student$ pg_lsclusters
```

```
Ver Cluster Port Status Owner    Data directory              Log file
16  main    5432 online postgres /var/lib/postgresql/16/main
/var/log/postgresql/postgresql-16-main.log
```

Журнал сообщений

В журнал записываются

- служебные сообщения сервера
- сообщения пользовательских сеансов
- сообщения приложений

Настройка журнала

- расположение
- формат записей
- какие события регистрировать

Информация о ходе работы СУБД записывается в журнал сообщений сервера. Сюда попадают сведения о запуске и останове сервера, различная служебная информация, в том числе сообщения о возникающих проблемах.

Также сюда могут выводиться сообщения о выполняющихся командах и времени их работы, о возникающих блокировках и тому подобное. Это позволяет выполнять трассировку пользовательских сеансов.

Разработчики приложений могут формировать и записывать в журнал сервера свои собственные сообщения.

Настройки PostgreSQL позволяют гибко определять, какие именно сообщения и в каком формате должны попадать в журнал сервера.

Например, вывод в форматах csv и json удобен для автоматизации анализа журнала.

https://postgrespro.ru/docs/postgresql/16/runtime-config-logging

Журнал сообщений сервера

Журнал сообщений сервера находится здесь:

```
student$ ls -l /var/log/postgresql/postgresql-16-main.log
```

```
-rw-r----- 1 postgres adm 5832 июл  5 16:17 /var/log/postgresql/postgresql-16-main.log
```

Заглянем в конец журнала:

```
student$ tail -n 10 /var/log/postgresql/postgresql-16-main.log
```

```
2024-07-05 16:17:19.409 MSK [616] LOG:  background worker "logical replication launcher"
(PID 674) exited with exit code 1
2024-07-05 16:17:19.409 MSK [669] LOG:  shutting down
2024-07-05 16:17:19.419 MSK [669] LOG:  checkpoint starting: shutdown immediate
2024-07-05 16:17:19.489 MSK [669] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0
WAL file(s) added, 0 removed, 0 recycled; write=0.021 s, sync=0.010 s, total=0.081 s;
sync files=2, longest=0.005 s, average=0.005 s; distance=0 kB, estimate=0 kB;
lsn=0/1956878, redo lsn=0/1956878
2024-07-05 16:17:19.495 MSK [616] LOG:  database system is shut down
2024-07-05 16:17:20.100 MSK [3440] LOG:  starting PostgreSQL 16.3 (Ubuntu
16.3-1.pgdg22.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
11.4.0-1ubuntu1~22.04) 11.4.0, 64-bit
2024-07-05 16:17:20.101 MSK [3440] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2024-07-05 16:17:20.114 MSK [3440] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2024-07-05 16:17:20.162 MSK [3443] LOG:  database system was shut down at 2024-07-03
15:57:35 MSK
2024-07-05 16:17:20.186 MSK [3440] LOG:  database system is ready to accept connections
```

Параметры конфигурации

Для всего экземпляра

- основной файл параметров — postgresql.conf
- `ALTER SYSTEM` — postgresql.auto.conf

Для текущего сеанса

- `SET/RESET`
- `set_config()`

Просмотр текущего значения

- `SHOW`
- `current_setting()`
- `pg_settings`

Сервер PostgreSQL настраивается с помощью разнообразных параметров конфигурации, которые позволяют управлять ресурсами, настраивать служебные процессы и пользовательские сеансы, управлять журналом сервера и решать многие другие задачи. Поэтому нужно знать, как проверить текущие значения параметров и установить новые.

Настройки всего сервера обычно задаются в конфигурационных файлах. Основной конфигурационный файл — postgresql.conf, он редактируется вручную. Второй конфигурационный файл — postgresql.auto.conf — предназначен для изменения специальной командой ALTER SYSTEM. Параметры, установленные через ALTER SYSTEM, имеют приоритет над параметрами в postgresql.conf.

Директивы включения файлов и каталогов include и include_dir позволяют разделять сложные файлы postgresql.conf на части. Это может быть удобно, например, при управлении несколькими серверами с похожими конфигурациями.

Большинство параметров конфигурации допускает изменение значений в пользовательских сеансах прямо во время выполнения. Помимо системных, можно определять и пользовательские параметры и работать с ними с помощью этих же команд и функций.

Варианты установки и управления параметрами:
https://postgrespro.ru/docs/postgresql/16/config-setting
Текущие значения параметров доступны в представлении pg_settings:
https://postgrespro.ru/docs/postgresql/16/view-pg-settings

Параметры конфигурации

Основной файл конфигурации postgresql.conf расположен в этом каталоге:

```
student$ ls -l /etc/postgresql/16/main
```

```
total 60
drwxr-xr-x 2 postgres postgres  4096 июл  3 15:57 conf.d
-rw-r--r-- 1 postgres postgres   315 июл  3 15:57 environment
-rw-r--r-- 1 postgres postgres   143 июл  3 15:57 pg_ctl.conf
-rw-r----- 1 postgres postgres  5743 июл  3 15:57 pg_hba.conf
-rw-r----- 1 postgres postgres  2640 июл  3 15:57 pg_ident.conf
-rw-r--r-- 1 postgres postgres 29960 июл  3 15:57 postgresql.conf
-rw-r--r-- 1 postgres postgres   317 июл  3 15:57 start.conf
```

Здесь же находятся и другие конфигурационные файлы.

Проверим значение параметра work_mem:

```
=> SHOW work_mem;
```

```
 work_mem
----------
 4MB
(1 row)
```

Параметр work_mem задает объем памяти, который будет использоваться для внутренних операций сортировки и размещения хеш-таблиц, прежде чем будут задействованы временные файлы на диске.

4MB — это значение по умолчанию и оно слишком мало. Допустим, мы хотим увеличить его до 16MB для всего экземпляра. Для этого есть различные пути.

Во-первых, можно внести изменение в postgresql.conf, раскомментировав и изменив строку, где определяется параметр:

```
student$ grep '#work_mem' /etc/postgresql/16/main/postgresql.conf
```

```
#work_mem = 4MB                         # min 64kB
```

Во-вторых, можно поместить определение параметра в файл с суффиксом .conf в каталоге /etc/postgresql/16/main/conf.d или в пользовательский файл конфигурации, местоположение которого следует задать в параметре include основного конфигурационного файла postgresql.conf.

В-третьих, можно изменить значение параметра с помощью команды SQL — что мы и сделаем:

```
=> ALTER SYSTEM SET work_mem TO '16MB';
```

```
ALTER SYSTEM
```

Такое изменение попадает не в postgresql.conf, а в файл postgresql.auto.conf, который находится в каталоге PGDATA:

```
student$ sudo cat /var/lib/postgresql/16/main/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
work_mem = '16MB'
```

Чтобы изменение вступило в силу, нужно перечитать конфигурационные файлы. Для этого можно воспользоваться pg_ctlcluster, либо использовать функцию SQL:

```
=> SELECT pg_reload_conf();
```

```
 pg_reload_conf
----------------
 t
(1 row)
```

Убедимся, что новое значение параметра применилось. Кроме команды SHOW, можно сделать это таким образом:

```
=> SELECT current_setting('work_mem');
```

```
 current_setting
-----------------
 16MB
(1 row)
```

Чтобы восстановить значение параметра по умолчанию, достаточно вместо SET использовать команду RESET (и, конечно, перечитать конфигурационные файлы):

```
=> ALTER SYSTEM RESET work_mem;

ALTER SYSTEM

=> SELECT pg_reload_conf();

 pg_reload_conf
----------------
 t
(1 row)
```

Большинству параметров можно установить новое значение для текущего сеанса прямо во время его выполнения. Например, если мы собираемся выполнить запрос, сортирующий большой объем данных, то для сеанса можно увеличить значение work_mem:

```
=> SET work_mem = '64MB';

SET
```

Новое значение действует только в текущем сеансе или даже в текущей транзакции (при указании SET LOCAL).

Еще один способ проверить текущее значение — выполнить запрос к представлению:

```
=> SELECT name, setting, unit FROM pg_settings WHERE name = 'work_mem';

   name   | setting | unit
----------+---------+------
 work_mem | 65536   | kB
(1 row)
```

Можно увидеть значение параметра и с помощью команды \dconfig:

```
=> \dconfig work_mem

List of configuration parameters
 Parameter | Value
-----------+-------
 work_mem  | 64MB
(1 row)
```

Клиент psql

Терминальный клиент для работы с PostgreSQL

Поставляется вместе с СУБД

Используется администраторами и разработчиками для интерактивной работы и выполнения скриптов

Для работы с СУБД PostgreSQL существуют различные сторонние инструменты, рассмотрение которых не входит в рамки курса.

В курсе мы будем использовать терминальный клиент psql:

1. psql — это единственный клиент, поставляемый вместе с СУБД.

2. Навыки работы с psql пригодятся разработчикам и администраторам вне зависимости от того, с каким инструментом они будут работать.

Для интерактивной работы в psql встроена поддержка readline, программ постраничного просмотра результатов запросов (таких, как less и pspg), а также подключения внешних редакторов. Возможности psql позволяют взаимодействовать с ОС, просматривать содержимое системного каталога, создавать скрипты для автоматизации повторяющихся задач.

https://postgrespro.ru/docs/postgresql/16/app-psql

Подключение

При запуске psql нужно указать параметры подключения. К обязательным параметрам относятся:

- имя базы данных, по умолчанию совпадает с именем пользователя;
- имя пользователя (роль), по умолчанию совпадает с именем пользователя ОС;
- узел (host), по умолчанию — локальное соединение;
- порт, по умолчанию — обычно 5432.

Параметры указываются так:

```
student$ psql -d база -U роль -h узел -p порт
```

Настройки, сделанные в виртуальной машине курса, позволяют подключаться к PostgreSQL без указания параметров:

```
student$ psql
```

Проверим текущее подключение:

```
=> \conninfo
```

```
You are connected to database "student" as user "student" via socket in
"/var/run/postgresql" at port "5432".
```

Команда \connect выполняет новое подключение, не покидая psql. Ее можно сократить до \c. Мы будем указывать необязательную часть имени команды в квадратных скобках: \c[onnect].

Справочная информация

Справку по psql можно получить не только в документации, но и прямо в системе. Команда

```
student$ psql --help
```

выдает справку по запуску. А если PostgreSQL устанавливался с документацией, то справочное руководство можно получить командой

```
student$ man psql
```

Утилита psql умеет выполнять команды SQL и свои собственные команды, которые начинаются с обратной косой черты, как \conninfo. Команды psql всегда однострочные — в отличие от команд SQL.

Внутри psql есть возможность получить список и краткое описание его собственных команд:

- \? выдает список команд psql,
- \h[elp] выдает список команд SQL, которые поддерживает сервер, а также синтаксис конкретной команды SQL.

Форматирование вывода

Клиент psql умеет выводить результаты запросов в разных форматах:

- формат с выравниванием значений;
- формат без выравнивания;
- расширенный формат.

Формат с выравниванием используется по умолчанию:

```
=> SELECT name, setting, unit FROM pg_settings LIMIT 7;
```

```
            name            |  setting   | unit
----------------------------+------------+------
 allow_in_place_tablespaces | off        |
 allow_system_table_mods    | off        |
 application_name           | psql       |
 archive_cleanup_command    |            |
 archive_command            | (disabled) |
 archive_library            |            |
 archive_mode               | off        |
(7 rows)
```

Ширина столбцов выровнена по значениям. Также выводится строка заголовков и итоговая строка.

Команды psql для переключения режима выравнивания:

- \a — переключатель режима: с выравниванием/без выравнивания.
- \t — переключатель отображения строки заголовка и итоговой строки.

Отключим выравнивание, заголовок и итоговую строку:

```
=> \a \t
```

```
Output format is unaligned.
Tuples only is on.
```

```
=> SELECT name, setting, unit FROM pg_settings LIMIT 7;
```

```
allow_in_place_tablespaces|off|
allow_system_table_mods|off|
application_name|psql|
archive_cleanup_command||
archive_command|(disabled)|
archive_library||
archive_mode|off|
```

```
=> \a \t
```

```
Output format is aligned.
Tuples only is off.
```

Такой формат неудобен для просмотра, но может оказаться полезным для автоматической обработки результатов.

Расширенный формат удобен, когда нужно вывести много столбцов для одной или нескольких записей. Для этого вместо точки с запятой указываем в конце команды \gx:

```
=> SELECT name, setting, unit, category, context, vartype,
    min_val, max_val, boot_val, reset_val
    FROM pg_settings
    WHERE name = 'work_mem' \gx
```

```
-[ RECORD 1 ]----------------------
name      | work_mem
setting   | 4096
unit      | kB
category  | Resource Usage / Memory
context   | user
vartype   | integer
min_val   | 64
max_val   | 2147483647
boot_val  | 4096
reset_val | 4096
```

Если расширенный формат нужен не для одной команды, а постоянно, можно включить его переключателем \x. Все возможности форматирования результатов запросов доступны через команду \pset.

Взаимодействие с ОС и выполнение скриптов

Из psql можно выполнять команды shell:

```
=> \! pwd
```

```
/home/student
```

С помощью запроса SQL можно сформировать несколько других запросов SQL и записать их в файл, используя команду \o[ut]:

```
=> \a \t \pset fieldsep ''
```

```
Output format is unaligned.
Tuples only is on.
Field separator is "".
```

```
=> \o dev1_tools.sql
```

```
=> SELECT format('SELECT %L AS tbl, count(*) FROM %I;', tablename, tablename)
FROM pg_tables LIMIT 3;
```

На экран (в стандартный вывод) ничего не попало. Посмотрим в файле:

```
=> \! cat dev1_tools.sql
```

```
SELECT 'pg_statistic' AS tbl, count(*) FROM pg_statistic;
SELECT 'pg_type' AS tbl, count(*) FROM pg_type;
SELECT 'pg_foreign_table' AS tbl, count(*) FROM pg_foreign_table;
```

Вернем вывод на экран и восстановим форматирование по умолчанию.

```
=> \o \t \a
```

```
Tuples only is off.
Output format is aligned.
```

Выполним теперь эти команды из файла с помощью \i[nclude]:

```
=> \i dev1_tools.sql
```

```
     tbl      | count
--------------+-------
 pg_statistic |   409
(1 row)

   tbl   | count
---------+-------
 pg_type |   613
(1 row)

       tbl        | count
------------------+-------
 pg_foreign_table |     0
(1 row)
```

Есть и другие способы выполнить команды, в том числе из файлов. После выполнения команд сеанс psql будет завершен:

- psql < имя_файла
- psql -f имя_файла
- psql -c 'команда' (работает только для одной команды)

Переменные psql

По аналогии с shell, psql имеет собственные переменные.

Установим переменную:

```
=> \set TEST Hi!
```

Чтобы подставить значение переменной, надо предварить ее имя двоеточием:

```
=> \echo :TEST
```

```
Hi!
```

Значение переменной можно сбросить:

```
=> \unset TEST
```

```
=> \echo :TEST
```

```
:TEST
```

Переменные можно использовать, например, для хранения текста часто используемых запросов. Вот запрос на получение списка пяти самых больших по размеру таблиц:

```
=> \set top5 'SELECT tablename, pg_total_relation_size(schemaname||''.''||tablename) AS bytes FROM pg_tables ORDER BY bytes DESC LIMIT 5;'
```

Для выполнения запроса достаточно набрать:

```
=> :top5
```

```
   tablename    |  bytes
----------------+---------
 pg_proc        | 1245184
 pg_rewrite     |  745472
 pg_attribute   |  720896
 pg_description |  630784
 pg_statistic   |  294912
(5 rows)
```

Присвоение значения переменной top5 удобно записать в стартовый файл .psqlrc в домашнем каталоге пользователя. Команды из .psqlrc будут автоматически выполняться каждый раз при старте psql.

Без параметров \set выдает значения всех переменных, включая встроенные. Справку по встроенным переменным можно получить так:

```
\? variables
```

Выйти из psql можно с помощью команд \q[uit], quit, exit или нажав Ctrl+D:

```
=> \q
```

Архитектура

Общее устройство PostgreSQL

Темы

Клиент-серверный протокол

Транзакционность и механизмы ее реализации

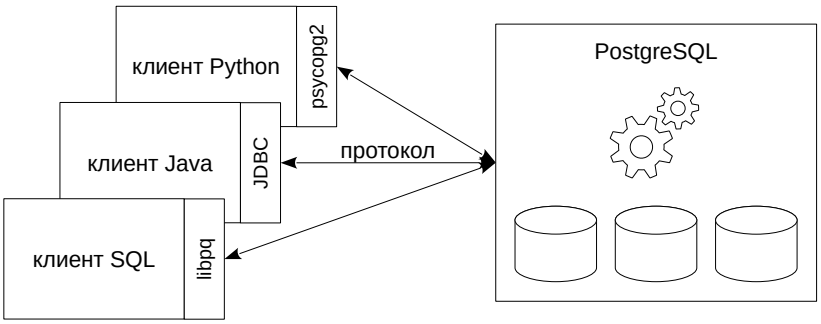
Схема обработки и способы выполнения запросов

Процессы и структуры памяти

Хранение данных на диске и работа с ними

Расширяемость системы

Клиент и сервер



подключение
формирование запросов
управление транзакциями

аутентификация
выполнение запросов
поддержка транзакционности

Клиентское приложение — например, psql или любая другая программа, написанная на любом языке программирования, — подключается к серверу и «общается» с ним. Чтобы клиент и сервер понимали друг друга, они должны использовать один и тот же *протокол* взаимодействия. Обычно клиент использует *драйвер,* реализующий протокол и предоставляющий набор функций для использования в программе. Внутри драйвер может пользоваться стандартной реализацией протокола (библиотекой libpq), либо реализовывать этот протокол самостоятельно.

Не так важно, на каком языке написан клиент — за разным синтаксисом будут стоять возможности, определенные протоколом. Мы будем использовать для примеров язык SQL с помощью клиента psql.
В реальной жизни клиентскую часть на SQL пишут редко, для учебных целей но нам это удобно. Мы рассчитываем, что сопоставить команды SQL с аналогичными возможностями какого-либо другого языка программирования не составит для вас большого труда.

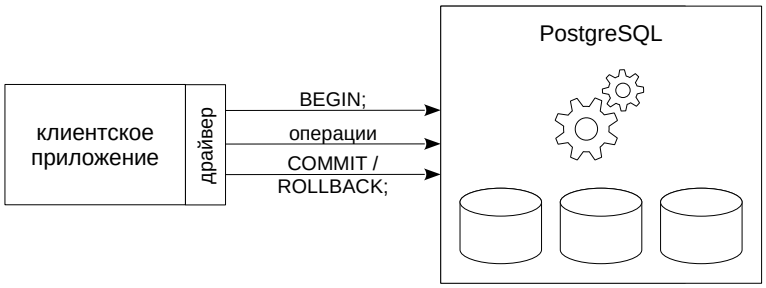
Протокол позволяет клиенту подключиться к одной из баз данных кластера. При этом сервер выполняет *аутентификацию* — решает, можно ли разрешить подключение, например, запросив пароль.

Далее клиент посылает серверу запросы на языке SQL, а сервер выполняет их и возвращает результат. Наличие мощного и удобного языка запросов — одна из особенностей реляционных СУБД.

Другая особенность — поддержка согласованной работы транзакций.

https://postgrespro.ru/docs/postgresql/16/protocol

Транзакции



атомарность — *все или ничего*
согласованность — *ограничения целостности и пользовательские ограничения*
изоляция — *влияние параллельных процессов*
долговечность — *сохранность данных даже после сбоя*

Под *транзакцией* понимается последовательность операций, которая сохраняет согласованность данных при условии, что операции выполнены полностью и без помех со стороны других транзакций.

От транзакций ожидают выполнения четырех свойств (ACID):

- Атомарность: транзакция либо выполняется полностью, либо не выполняется вовсе. Для этого начало транзакции отмечается командой BEGIN, а конец — либо COMMIT (фиксация изменений), либо ROLLBACK (отмена изменений).
- Согласованность: транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние (согласованность - соблюдение установленных ограничений).
- Изоляция: другие транзакции, выполняющиеся одновременно с данной, не должны оказывать на нее влияния.
- Долговечность: после того, как данные зафиксированы, они не должны потеряться даже в случае сбоя.

За управление транзакциями (то есть за определение команд, составляющих транзакцию, и за фиксацию или отмену транзакции) в PostgreSQL, как правило, отвечает клиентское приложение. На стороне сервера управлять транзакциями могут хранимые процедуры.

https://postgrespro.ru/docs/postgresql/16/sql-begin

https://postgrespro.ru/docs/postgresql/16/sql-savepoint

https://postgrespro.ru/docs/postgresql/16/transactions

Управление транзакциями

По умолчанию psql работает в режиме автофиксации:

```
=> \echo :AUTOCOMMIT
```

```
on
```

Это приводит к тому, что любая одиночная команда, выданная без явного указания начала транзакции, сразу же фиксируется.

- Проверьте, включен ли аналогичный режим в драйвере PostgreSQL вашего любимого языка программирования?

Создадим таблицу с одной строкой:

```
=> CREATE TABLE t(
  id integer,
  s text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(id, s) VALUES (1, 'foo');
```

```
INSERT 0 1
```

Увидит ли таблицу и строку другая транзакция?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

Да. Сравните:

```
=> BEGIN; -- явно начинаем транзакцию
```

```
BEGIN
```

```
=> INSERT INTO t(id, s) VALUES (2, 'bar');
```

```
INSERT 0 1
```

Что увидит другая транзакция на этот раз?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

Изменения еще не зафиксированы, поэтому не видны другой транзакции.

```
=> COMMIT;
```

```
COMMIT
```

А теперь?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
(2 rows)
```

Режим без автофиксации неявно начинает транзакцию при первой выданной команде; изменения надо фиксировать самостоятельно.

```
=> \set AUTOCOMMIT off
```

```
=> INSERT INTO t(id, s) VALUES (3, 'baz');
```

```
INSERT 0 1
```

Что на этот раз?

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
(2 rows)
```

Изменения не видны; транзакция была начата неявно.

```
=> COMMIT;
```

```
COMMIT
```

Ну и наконец:

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
(3 rows)
```

Восстановим режим, в котором psql работает по умолчанию.

```
=> \set AUTOCOMMIT on
```

Отдельные изменения можно откатывать, не прерывая транзакцию целиком (хотя необходимость в этом возникает нечасто).

```
=> BEGIN;
```

```
BEGIN
```

```
=> SAVEPOINT sp; -- точка сохранения
```

```
SAVEPOINT
```

```
=> INSERT INTO t(id, s) VALUES (4, 'qux');
```

```
INSERT 0 1
```

```
=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
  4 | qux
(4 rows)
```

Обратите внимание: свои собственные изменения транзакция видит, даже если они не зафиксированы.

Теперь откатим все до точки сохранения.

Откат к точке сохранения не подразумевает передачу управления (то есть не работает как GOTO); отменяются только те изменения состояния БД, которые были выполнены от момента установки точки до текущего момента.

```
=> ROLLBACK TO sp;
```

```
ROLLBACK
```

Что увидим?

```
=> SELECT * FROM t;
```

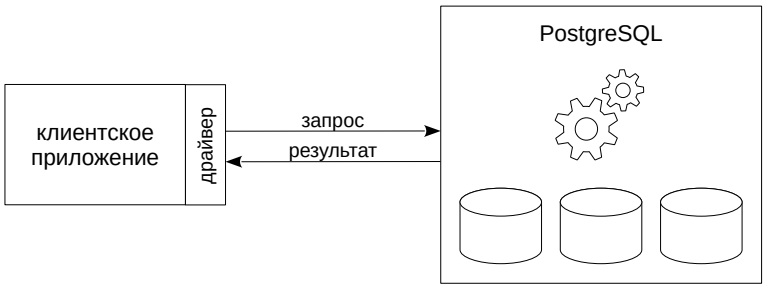
```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
(3 rows)
```

Сейчас изменения отменены, но транзакция продолжается:

```
=> INSERT INTO t(id, s) VALUES (4, 'xyz');
```

```
INSERT 0 1

=> COMMIT;

COMMIT

=> SELECT * FROM t;

 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
  4 | xyz
(4 rows)
```

Выполнение запроса



разбор	← *системный каталог*
переписывание	← *правила*
планирование	← *статистика*
выполнение	← *данные*

Выполнение запроса — довольно сложная задача. Запрос передается от клиента серверу в виде текста. Текст надо *разобрать* — выполнить синтаксический разбор (складываются ли буквы в слова, а слова — в команды) и семантический разбор (есть ли в базе данных таблицы и другие объекты, на которые запрос ссылается по имени). Для этого требуется информация о том, что вообще содержится в базе данных. Такая *мета*-информация называется *системным каталогом*, она хранится в самой базе данных в специальных таблицах.

Запрос может *переписываться* (трансформироваться) — например, вместо имени представления подставляется текст запроса. Можно придумать и свои трансформации, для чего есть механизм *правил.*

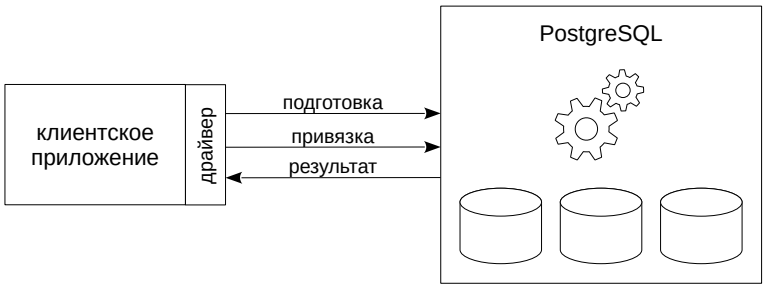
SQL — декларативный язык: запрос, составленный на нем, говорит о том, какие данные надо получить, но не говорит, как это сделать. Поэтому запрос (уже разобранный и представленный в виде дерева), передается *планировщику*, который разрабатывает *план выполнения*. Например, планировщик решает, надо ли использовать индексы. Чтобы качественно спланировать работу, планировщику нужна информация о размере таблиц и о распределении данных в них — *статистика*.

Далее запрос выполняется в соответствии с планом и результат возвращается клиенту — целиком и полностью:

https://postgrespro.ru/docs/postgresql/16/query-path

Это удобный и простой способ для небольших выборок, он обеспечивается *простым режимом* протокола.

Подготовка операторов



разбор
переписывание

привязка
планирование
выполнение

← *значения параметров*

Итак, каждый запрос проходит перечисленные ранее шаги: разбор, переписывание, планирование и выполнение. Но если один и тот же запрос (с точностью до параметров) выполняется много раз, нет смысла каждый раз разбирать его заново.

Поэтому кроме обычного выполнения запросов протокол PostgreSQL предусматривает *расширенный режим*, который позволяет более детально управлять выполнением операторов.

В качестве одной из возможностей расширенный режим позволяет *подготовить* запрос — заранее выполнить разбор и переписывание и запомнить дерево разбора.

При выполнении запроса выполняется *привязка* конкретных значений параметров. Если необходимо, здесь выполняется планирование (в некоторых случаях PostgreSQL запоминает план запроса и не выполняет повторно этот шаг). Затем запрос выполняется.

Еще одно преимущество использования подготовленных операторов — невозможность внедрения SQL-кода.

https://postgrespro.ru/docs/postgresql/16/sql-prepare

https://postgrespro.ru/docs/postgresql/16/sql-execute

Подготовленные операторы

В SQL оператор подготавливается командой PREPARE (эта команда является расширением PostgreSQL, она отсутствует в стандарте):

```
=> PREPARE q(integer) AS
  SELECT * FROM t WHERE id = $1;
```

```
PREPARE
```

При этом выполняются разбор и переписывание, и полученное дерево разбора запоминается.

После подготовки оператор можно вызывать по имени, передавая фактические параметры:

```
=> EXECUTE q(1);
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

Если у запроса нет параметров, при подготовке запоминается и построенный план выполнения. Если же параметры есть, как в этом примере, то их фактические значения принимаются во внимание при планировании. Планировщик может счесть, что план, построенный без учета параметров, окажется не хуже, и тогда перестанет выполнять планирование повторно.

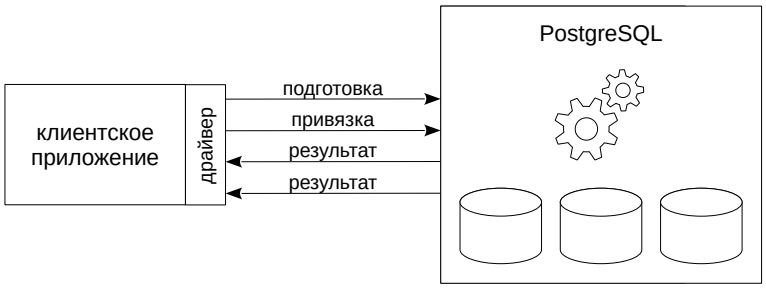
- А как подготовить и выполнить оператор в вашем любимом языке?
- Есть ли возможность выполнить оператор, НЕ подготавливая его?

Все подготовленные операторы текущего сеанса можно увидеть в представлении:

```
=> SELECT * FROM pg_prepared_statements \gx
```

```
-[ RECORD 1 ]---+---------------------------------
name            | q
statement       | PREPARE q(integer) AS           +
                |   SELECT * FROM t WHERE id = $1;
prepare_time    | 2024-07-05 16:17:57.488204+03
parameter_types | {integer}
result_types    | {integer,text}
from_sql        | t
generic_plans   | 0
custom_plans    | 1
```

Курсоры



разбор
переписывание

привязка ← *значения параметров*
планирование
выполнение

получение результата

Не всегда клиенту бывает удобно получить все результаты сразу. Данных может оказаться много, но не все они могут быть нужны.

Для этого расширенный режим протокола предусматривает *курсоры*. Можно открыть курсор для какого-либо оператора, а затем получать результирующие данные построчно по мере необходимости.

Курсор можно рассматривать как окно, в котором видна только часть из множества результатов. При получении строки данных окно сдвигается. Иными словами, курсоры позволяют работать с реляционными данными (множествами) итеративно, строка за строкой.

Открытый курсор представлен на сервере так называемым *порталом*. Это слово встречается в документации; в первом приближении можно считать «курсор» и «портал» синонимами.

Запрос, используемый в курсоре, неявно подготавливается (то есть сохраняется его дерево разбора и, возможно, план выполнения).

https://postgrespro.ru/docs/postgresql/16/sql-declare

https://postgrespro.ru/docs/postgresql/16/sql-fetch

Курсоры

При выполнении команды SELECT сервер передает, а клиент получает сразу все строки:

```
=> SELECT * FROM t ORDER BY id;
```

```
 id |  s
----+-----
  1 | foo
  2 | bar
  3 | baz
  4 | xyz
(4 rows)
```

Курсор позволяет получать данные построчно.

```
=> BEGIN;
```

```
BEGIN
```

```
=> DECLARE c CURSOR FOR
  SELECT * FROM t ORDER BY id;
```

```
DECLARE CURSOR
```

```
=> FETCH c;
```

```
 id |  s
----+-----
  1 | foo
(1 row)
```

Размер выборки можно указывать:

```
=> FETCH 2 c;
```

```
 id |  s
----+-----
  2 | bar
  3 | baz
(2 rows)
```

Этот размер играет важную роль, когда строк очень много: обрабатывать большой объем данных построчно крайне неэффективно.

Что, если в процессе чтения мы дойдем до конца таблицы?

```
=> FETCH 2 c;
```

```
 id |  s
----+-----
  4 | xyz
(1 row)
```

```
=> FETCH 2 c;
```

```
 id | s
----+---
(0 rows)
```

FETCH просто перестанет возвращать строки. В обычных языках программирования всегда есть возможность проверить это условие.

- Как в вашем языке программирования получать данные построчно с помощью курсора?
- Есть ли возможность НЕ пользоваться курсором и получить все строки сразу?
- Как настраивается размер выборки для курсора?

По окончании работы открытый курсор закрывают, освобождая ресурсы:

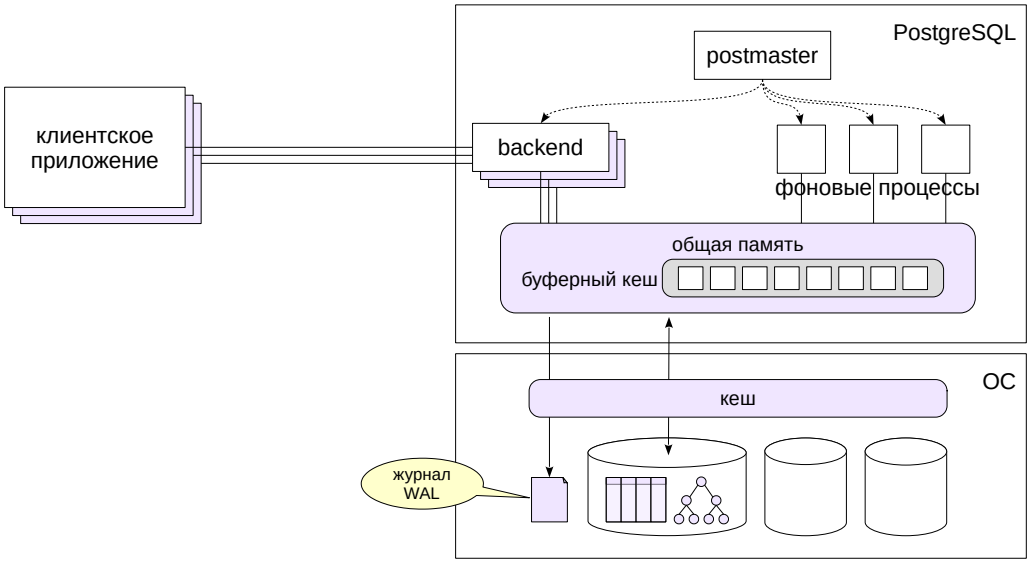
```
=> CLOSE c;
```

```
CLOSE CURSOR
```

Однако курсоры автоматически закрываются по завершению транзакции, так что можно не закрывать их явно.(Исключение составляют курсоры, открытые с указанием WITH HOLD.)

=> COMMIT;
COMMIT

Хранение данных

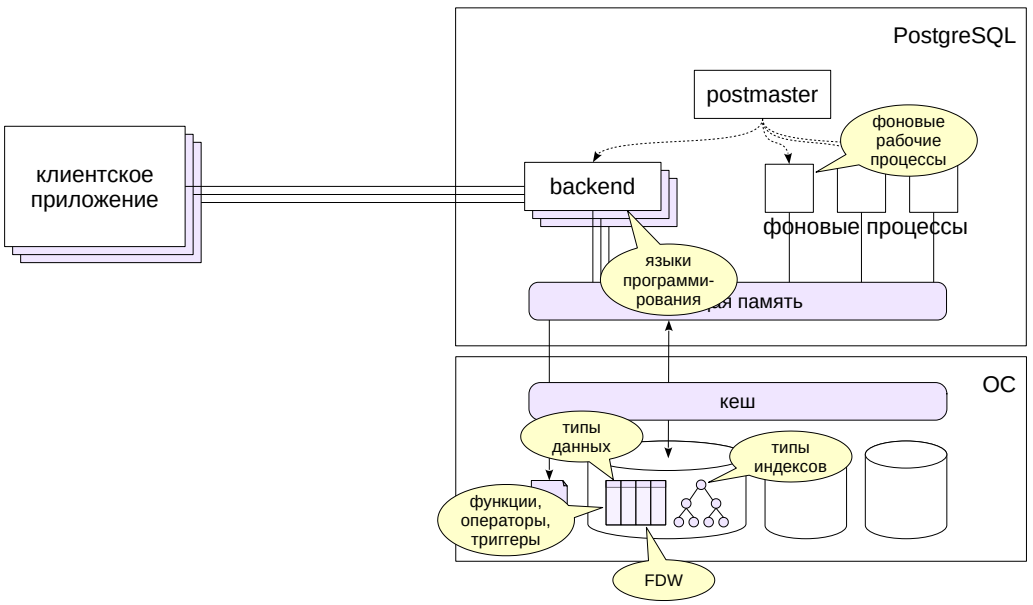


Буферный кеш PostgreSQL располагается в общей памяти, чтобы все процессы имели к нему доступ.

PostgreSQL работает с дисками, на которых находятся данные, не напрямую, а через операционную систему. У операционной системы тоже имеется собственный кеш данных. Поэтому, если страница не будет найдена в буферном кеше, остается шанс, что она есть в кеше ОС и обращения к диску удастся избежать.

При сбое (например, питания) содержимое оперативной памяти пропадает, при этом измененные, но еще не записанные на диск данные теряются. Это недопустимо и противоречит свойству долговечности транзакций. Поэтому в процессе работы PostgreSQL постоянно записывает специальный журнал, позволяющий повторно выполнить потерянные операции и восстановить данные в согласованном состоянии. Про буферный кеш и журнал мы будем говорить отдельно в одноименной теме.

Расширяемость



PostgreSQL спроектирован с расчетом на расширяемость. Он предоставляет возможность создавать новые типы данных на основе уже имеющихся, писать хранимые подпрограммы для обработки данных, а также обеспечивает удобный инструментарий для администрирования, мониторинга и настройки производительности.

При необходимости можно написать расширение, которое добавляет недостающий функционал. Большинство расширений можно устанавливать «на лету», без рестарта сервера. Благодаря такой архитектуре, существует большое количество расширений, которые:

- добавляют поддержку языков программирования (помимо стандартных SQL, PL/pgSQL, PL/Perl, PL/Python и PL/Tcl);
- вводят новые типы данных и операторы для работы с ними;
- создают новые типы индексов для эффективной работы с разнообразными типами данных (помимо стандартных B-деревьев, GiST, SP-GiST, GIN, BRIN, Bloom);
- запускают служебные фоновые процессы для выполнения дополнительных задач;
- позволяют подключаться к внешним источникам данных;
- собирают информацию о нагрузке на систему, выполняют мониторинг и строят отчеты;
- позволяют исследовать системные структуры данных.

Подробнее расширения рассматриваются в курсах DBA2 и DEV2.

Итоги

Сервер управляет кластером баз данных

Протокол позволяет клиентам подключаться к серверу, выполнять запросы и управлять транзакциями

Каждый клиент обслуживается своим процессом

Данные хранятся в файлах, обращение происходит через операционную систему

Кеширование как в локальной памяти (каталог, разобранные запросы), так и в общей (буферный кеш)