

Архитектура Общее устройство PostgreSQL



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Клиент-серверный протокол

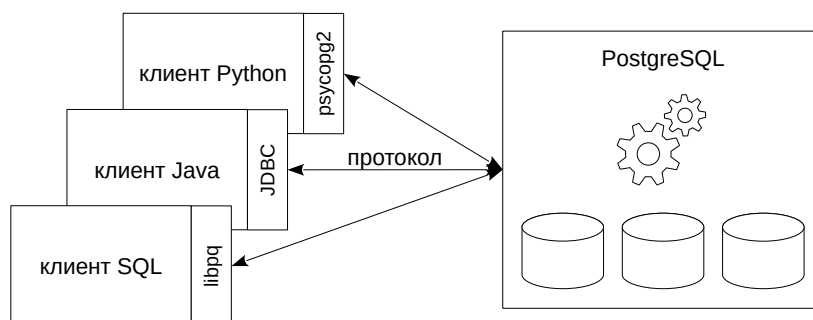
Транзакционность и механизмы ее реализации

Схема обработки и способы выполнения запросов

Процессы и структуры памяти

Хранение данных на диске и работа с ними

Расширяемость системы



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.

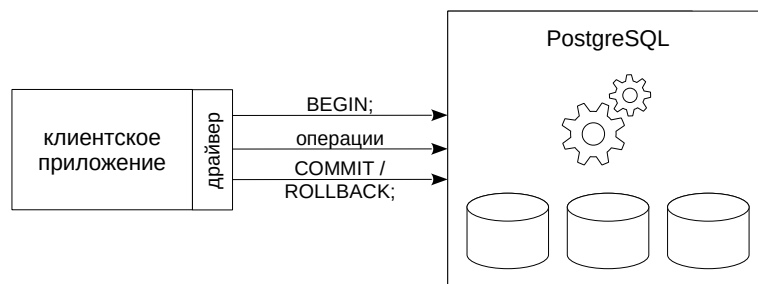
Не так важно, на каком языке написан клиент — за разным синтаксисом будут стоять возможности, определенные протоколом. Мы будем использовать для примеров язык SQL с помощью клиента `psql`. В реальной жизни клиентскую часть на SQL пишут редко, но нам это удобно для учебных целей. Мы рассчитываем, что 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>



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
```

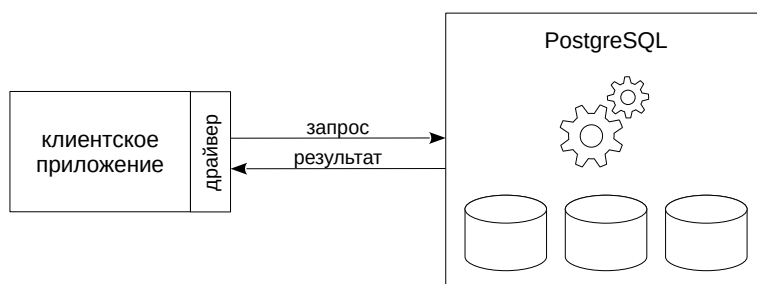
```
=> COMMIT;
```

```
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*.

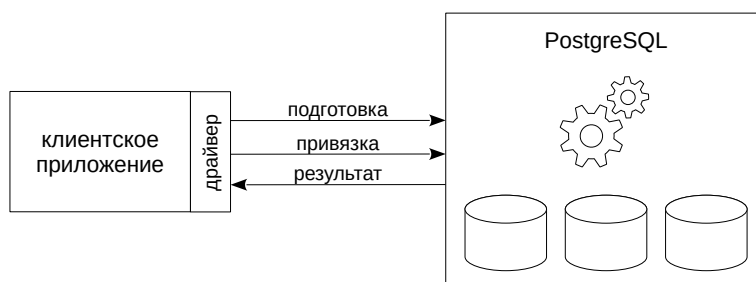
SQL — декларативный язык: запрос, составленный на нем, говорит о том, какие данные надо получить, но не говорит, как это сделать. Поэтому запрос (уже разобранный и представленный в виде дерева), передается *планировщику*, который разрабатывает *план выполнения*. Например, планировщик решает, надо ли 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)
```

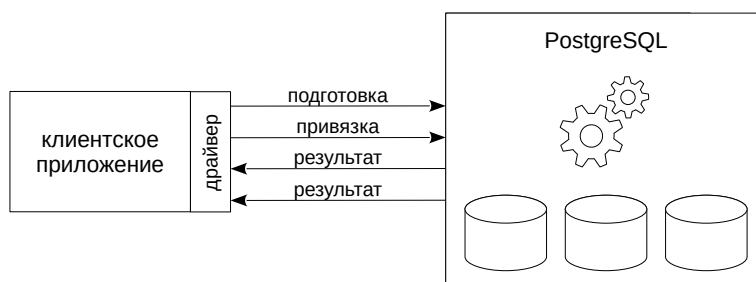
Если у запроса нет параметров, при подготовке запоминается и построенный план выполнения. Если же параметры есть, как в этом примере, то их фактические значения принимаются во внимание при планировании. Планировщик может счесть, что план, построенный без учета параметров, окажется не хуже, и тогда перестанет 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  | 2024-10-13 19:46:45.562993+03
parameter_types | {integer}
result_types  | {integer,text}
from_sql      | t
generic_plans | 0
custom_plans  | 1
```



разбор
переписывание

привязка
планирование
выполнение

получение результата

← значения параметров

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:

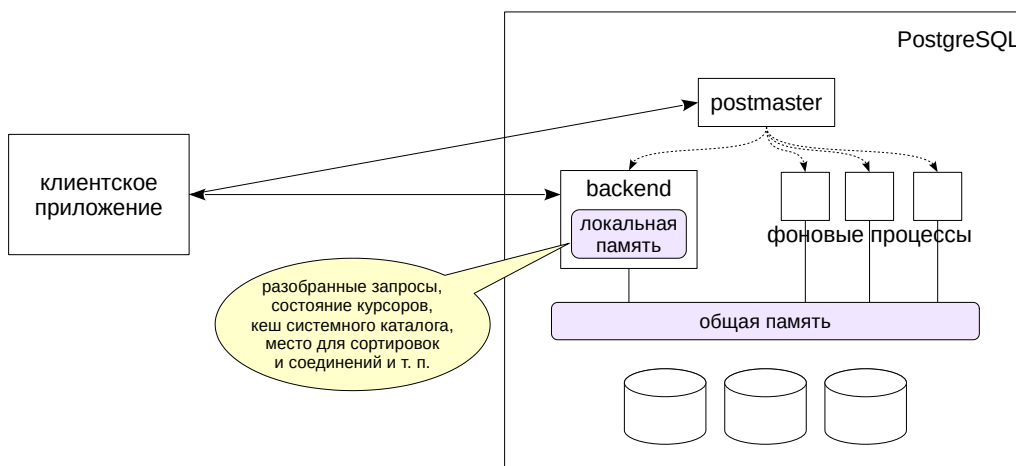
```
=> CLOSE c;
```

```
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
```



Между обращениями клиента сервер должен хранить вспомогательную информацию: разобранные запросы и их планы, состояние открытых курсоров (порталы). Где и как он это делает?

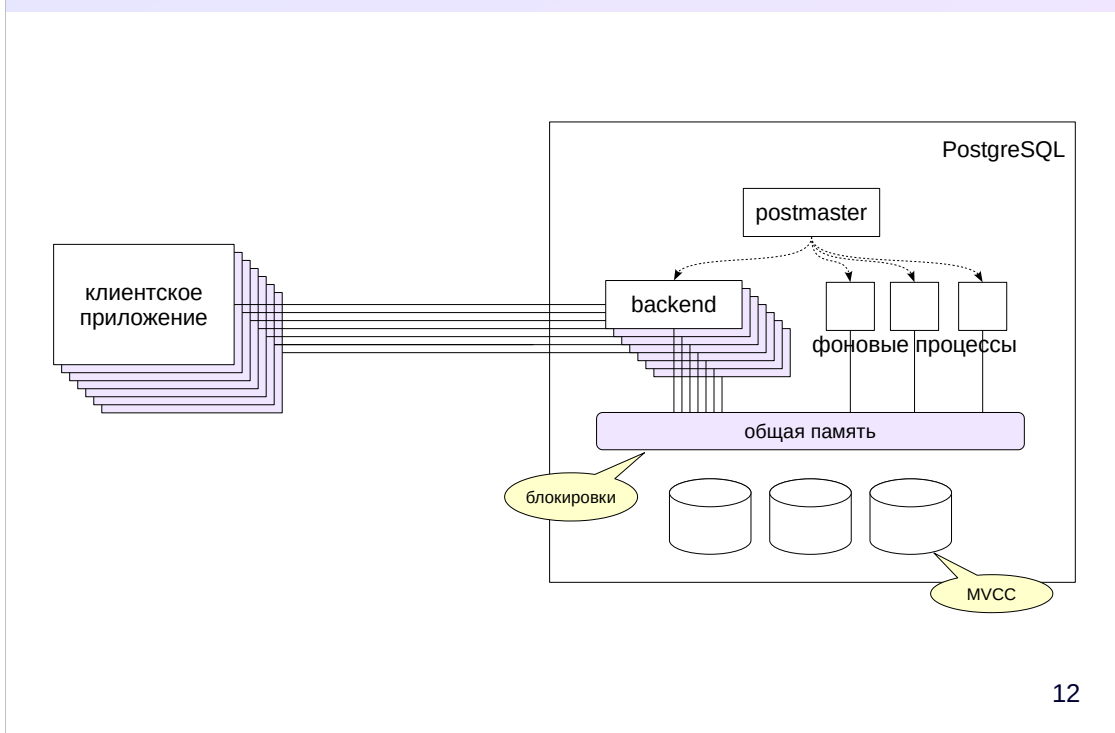
Сервер PostgreSQL состоит из нескольких взаимодействующих процессов. В первую очередь при старте сервера запускается процесс, традиционно называемый `postmaster`. Он запускает все остальные процессы (с помощью системного вызова `fork` в Unix) и «присматривает» за ними — если какой-нибудь процесс завершится аварийно, `postmaster` перезапустит его (или перезапустит весь сервер, если сочтет, что процесс мог повредить общие данные).

Работу сервера обеспечивает ряд фоновых служебных процессов. В следующих темах этого модуля мы поговорим об основных из них.

Чтобы процессы могли обмениваться информацией, `postmaster` выделяет *общую память*, доступ к которой могут получить все процессы. Кроме общей памяти, каждый процесс имеет и свою *локальную память*, доступную только ему самому.

`Postmaster` слушает входящие соединения. Для каждого вновь подключающегося клиента `postmaster` порождает обслуживающий процесс (`backend`) и дальше клиент общается уже с этим процессом. Обслуживающий процесс, в том числе, производит аутентификацию.

Место, необходимое для выполнения запроса (разобранные запросы и их планы, состояние курсоров, кеш системного каталога, место для сортировки данных и т. п.), выделяется в *локальной памяти* обслуживающего процесса.



Когда к серверу подключается много клиентов, для каждого из них порождается собственный обслуживающий процесс. Это не проблема, пока клиентов не очень много: на всех хватает оперативной памяти, а соединения не происходят слишком часто.

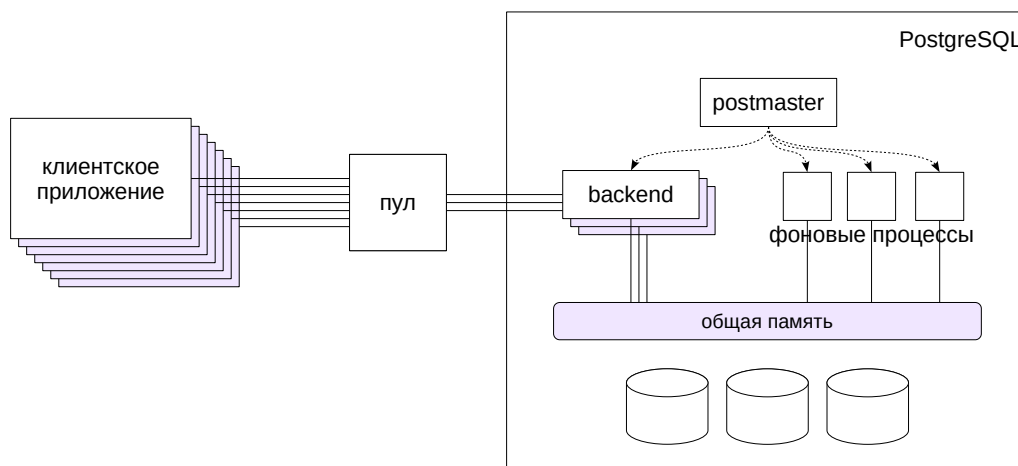
Тем не менее при одновременной работе с какими-либо объектами приходится принимать меры, чтобы один процесс не поменял какие-то данные в то время, пока с ними работает другой процесс.

Для объектов в общей памяти используются короткоживущие блокировки. PostgreSQL делает это достаточно аккуратно для того, чтобы система хорошо масштабировалась при увеличении числа процессоров (ядер).

С таблицами сложнее, поскольку блокировки нужно удерживать до конца транзакций (то есть потенциально в течение длительного времени), из-за чего масштабируемость может пострадать. Поэтому PostgreSQL использует механизм *многоверсионности* (MVCC, *multiversion concurrency control*) и *изоляция на основе снимков данных*: одни и те же данные могут одновременно существовать в разных версиях, а каждый процесс видит собственную (но всегда согласованную) картину данных. Это позволяет блокировать только те процессы, которые пытаются изменить данные, уже измененные, но еще не зафиксированные другими процессами.

Многоверсионность — основной механизм, который обеспечивает первые три свойства транзакций (атомарность, согласованность, изоляция). Про него мы будем говорить отдельно в теме «Изоляция и многоверсионность».

Пул соединений



13

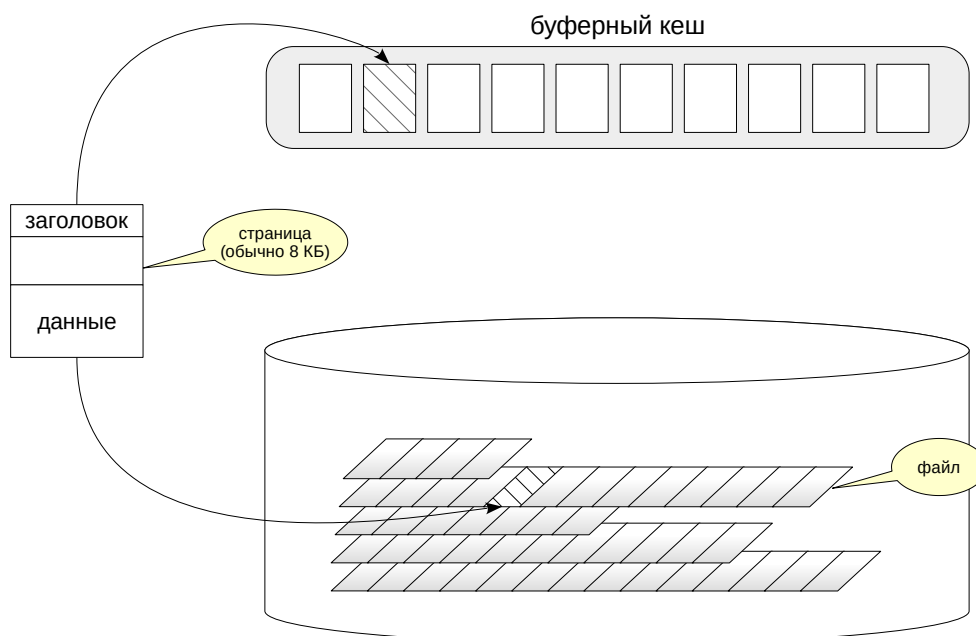
Если клиентов слишком много, или соединения устанавливаются и разрываются слишком часто, стоит подумать о применении пула соединений. Такую функцию обычно предоставляет сервер приложений или можно воспользоваться сторонними менеджерами пула (наиболее известен pgBouncer).

Клиенты подключаются не к серверу PostgreSQL, а к менеджеру пула. Менеджер удерживает открытыми несколько соединений с сервером баз данных и использует одно из свободных для того, чтобы выполнять запросы клиента. Таким образом, с точки зрения сервера число клиентов остается постоянным вне зависимости от того, сколько клиентов обращаются к менеджеру пула.

Но при таком режиме работы несколько клиентов разделяют один и тот же обслуживающий процесс, который — как мы говорили — в своей локальной памяти хранит определенное состояние (в частности, разобранные запросы для подготовленных операторов). Как правило, это необходимо учитывать при разработке приложений.

Одна из возможностей pgBouncer — временная приостановка обслуживания клиентов без разрыва соединения. Такая приостановка может использоваться для обновления программного обеспечения сервера или других операций, требующих рестарта сервера.

Подробнее вопросы применения пула соединений рассматриваются в курсе DEV2.

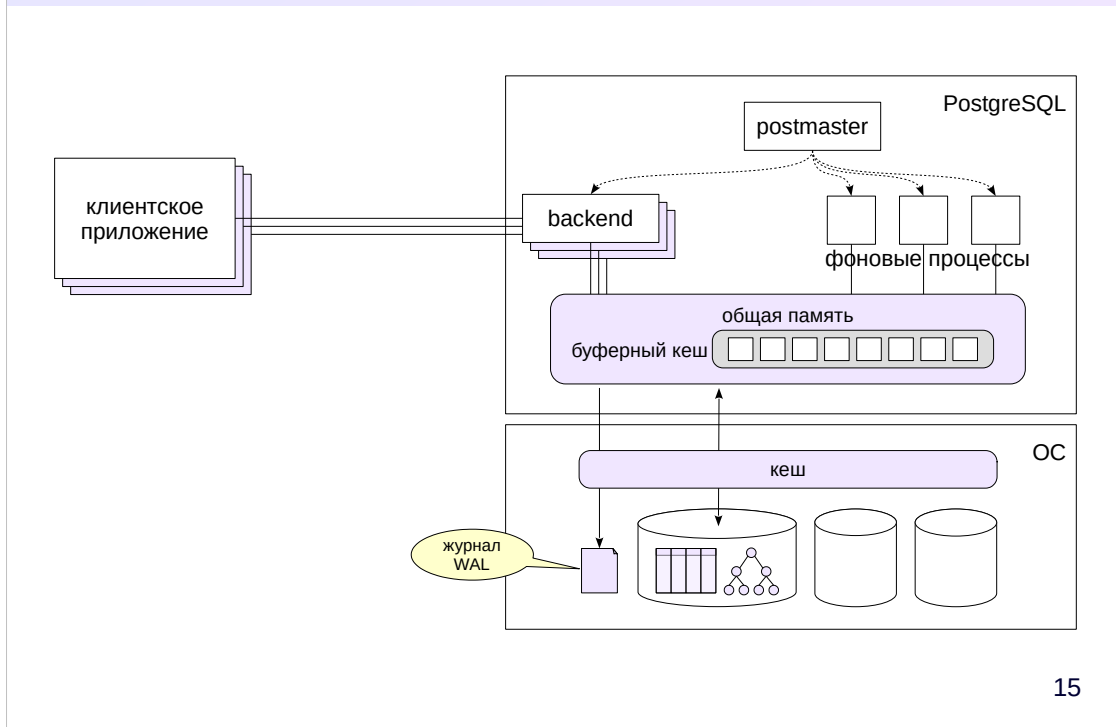


Данные хранятся на дисках в обычных файлах операционной системы. Как именно данные распределены по файлам, рассматривается в одной из следующих тем.

Логически файлы разделены на *страницы* (иногда используется термин *блок*). Страница обычно имеет размер 8 Кбайт. Его в некоторых пределах можно поменять (16 или 32 Кбайта), но только при сборке сервера. Таким образом, собранный и запущенный кластер может работать со страницами только одного размера.

Каждая страница имеет внутреннюю разметку. Страница содержит заголовок и полезные данные; между ними может находиться свободное пространство, если страница занята не полностью.

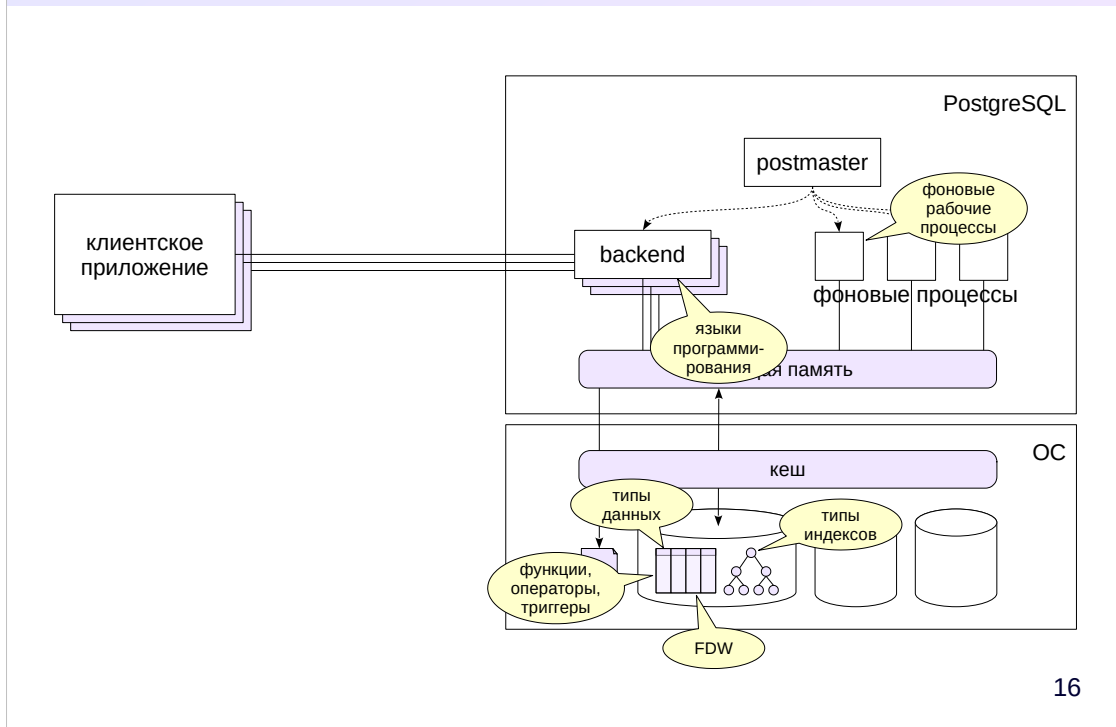
Из-за того, что диски работают значительно медленнее, чем оперативная память (особенно HDD, но и SSD тоже), применяется *кеширование*: в оперативной памяти отводится место под недавно прочитанные с диска страницы — *буферный кеш*. Расчет делается на то, что одни и те же страницы понадобятся несколько раз и можно будет сэкономить на повторном обращении к диску. Измененные данные также записываются на диск не сразу, а через некоторое время.



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.



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.

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)