Архитектура

Общее устройство PostgreSQL



Темы

Клиент-серверный протокол

Транзакционность и механизмы ее реализации

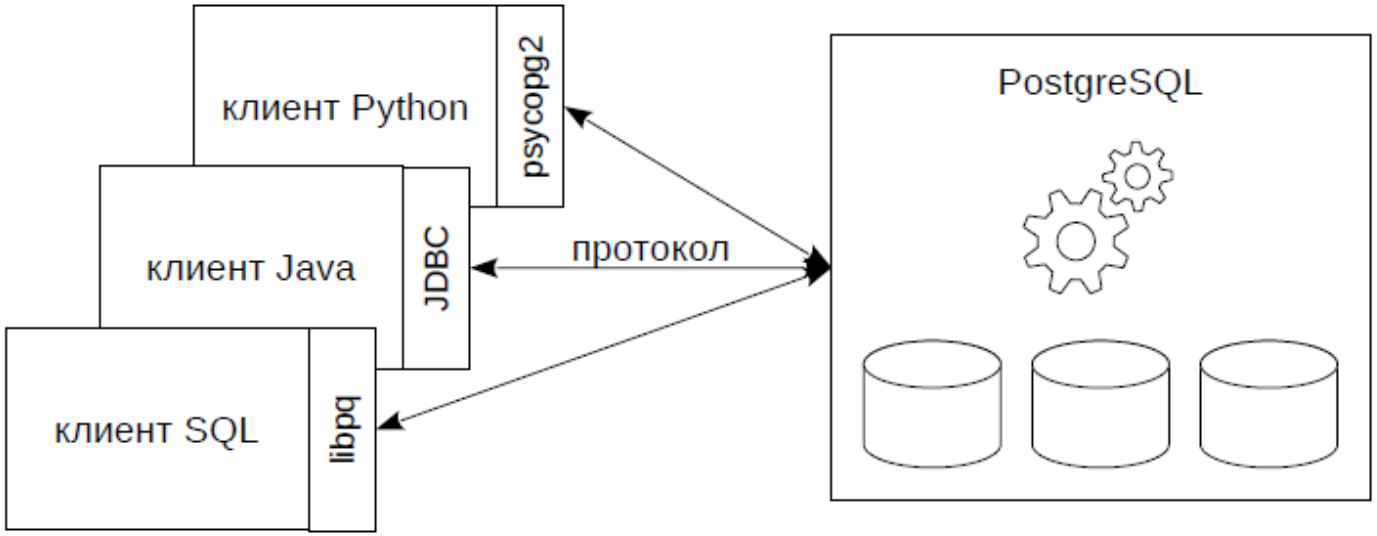
Схема обработки и способы выполнения запросов

Процессы и структуры памяти

Хранение данных на диске и работа с ними

Расширяемость системы

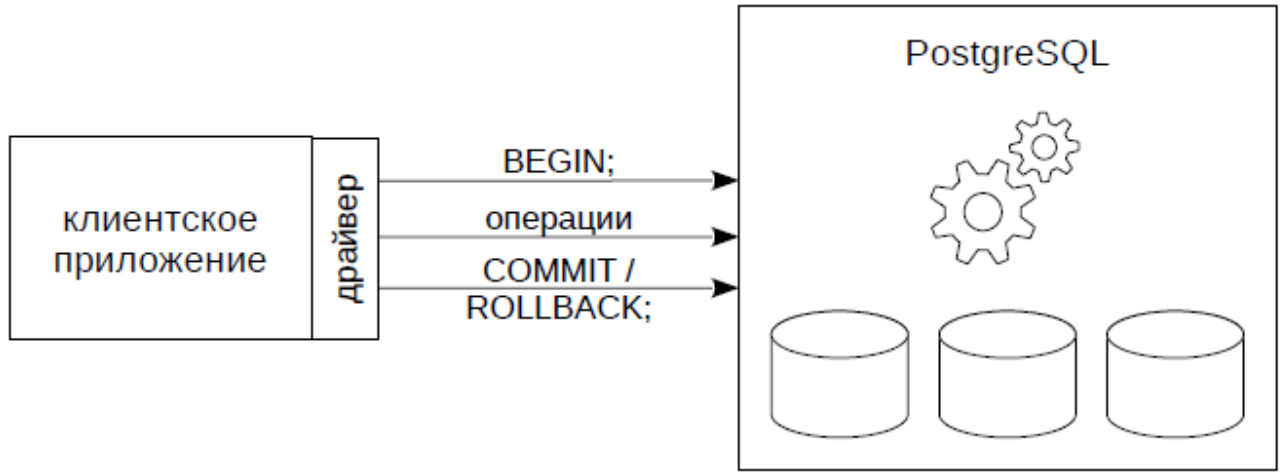
Клиент и сервер



подключение
формирование запросов
управление транзакциями

аутентификация
выполнение запросов
поддержка транзакционности

Транзакции



атомарность — *все или ничего*
согласованность — *ограничения целостности и пользовательские ограничения*
изоляция — *влияние параллельных процессов*

долговечность — *сохранность данных даже после сбоя*

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.

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)
```

```
=> 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
```

Восстановим режим, в котором psql работает по умолчанию.

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

```
=> 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)
```

```
=> 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

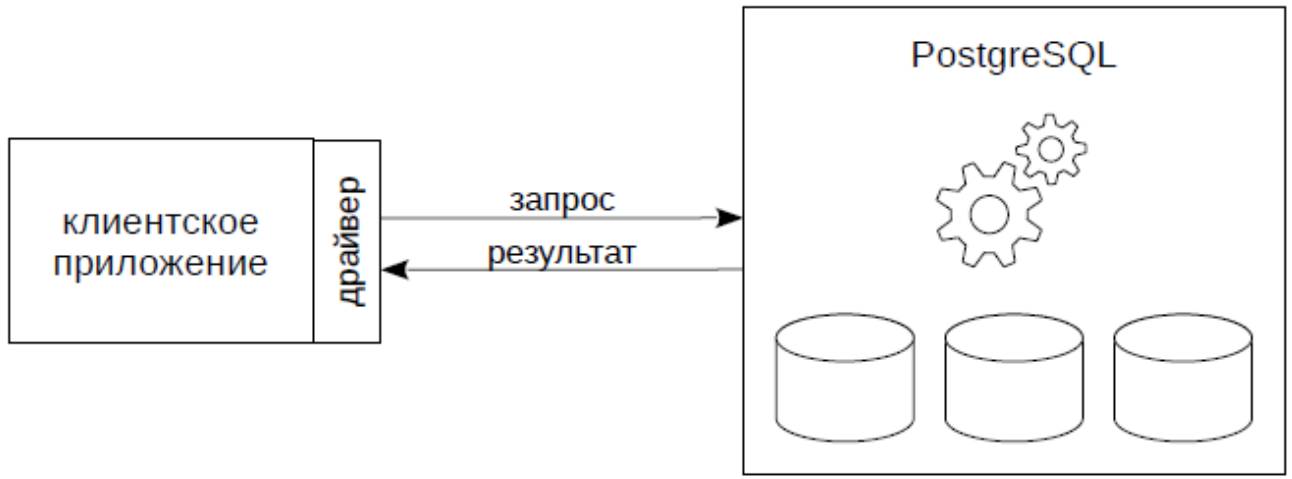
=> COMMIT;

COMMIT

=> SELECT * FROM t;

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

Выполнение запроса



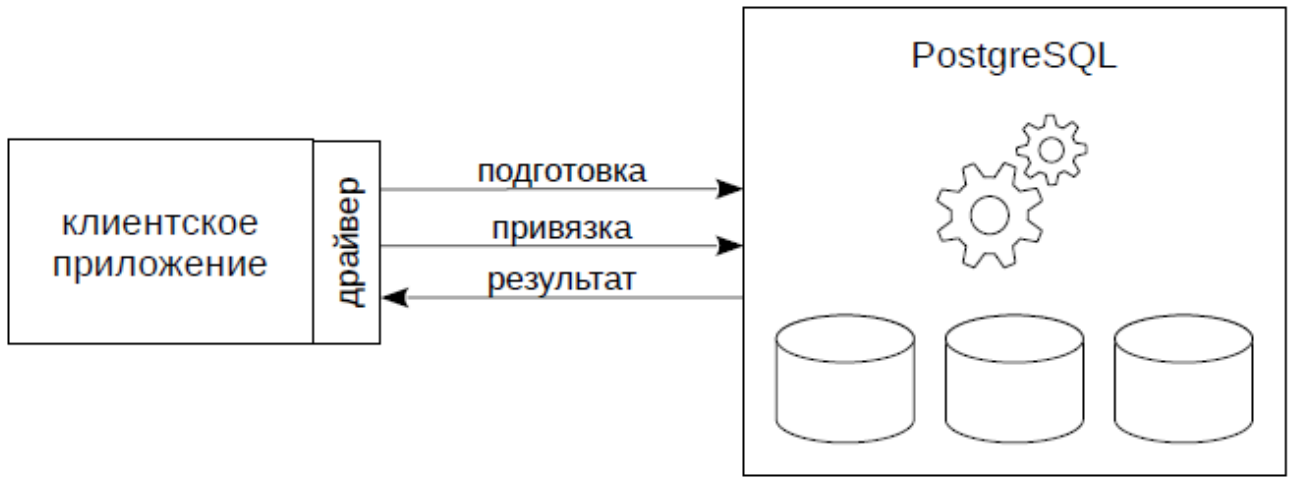
разбор ← *системный каталог*

переписывание ← *правила*

планирование ← *статистика*

выполнение ← *данные*

Подготовка операторов



разбор
переписывание

привязка ← *значения параметров*
планирование
выполнение

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)
```

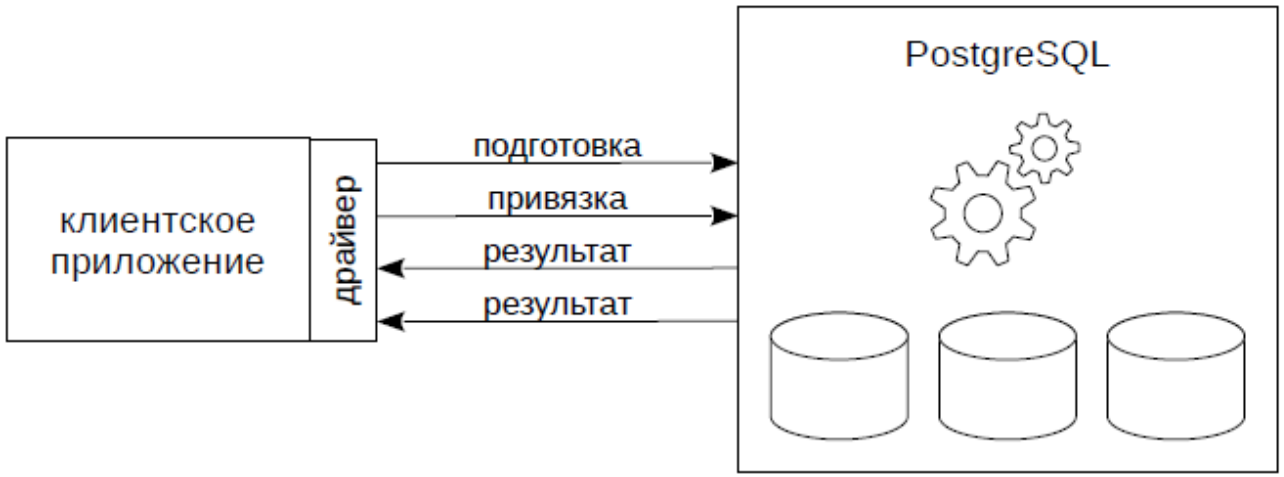
Если у запроса нет параметров, при подготовке запоминается и построенный план выполнения. Если же параметры есть, как в этом примере, то их фактические значения принимаются во внимание при планировании. Планировщик может счесть, что план, построенный без учета параметров, окажется не хуже, и тогда перестанет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

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    | 2025-06-24 04:29:41.922528+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

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.

По окончании работы открытый курсор закрывают, освобождая ресурсы:

```
=> CLOSE c;
```

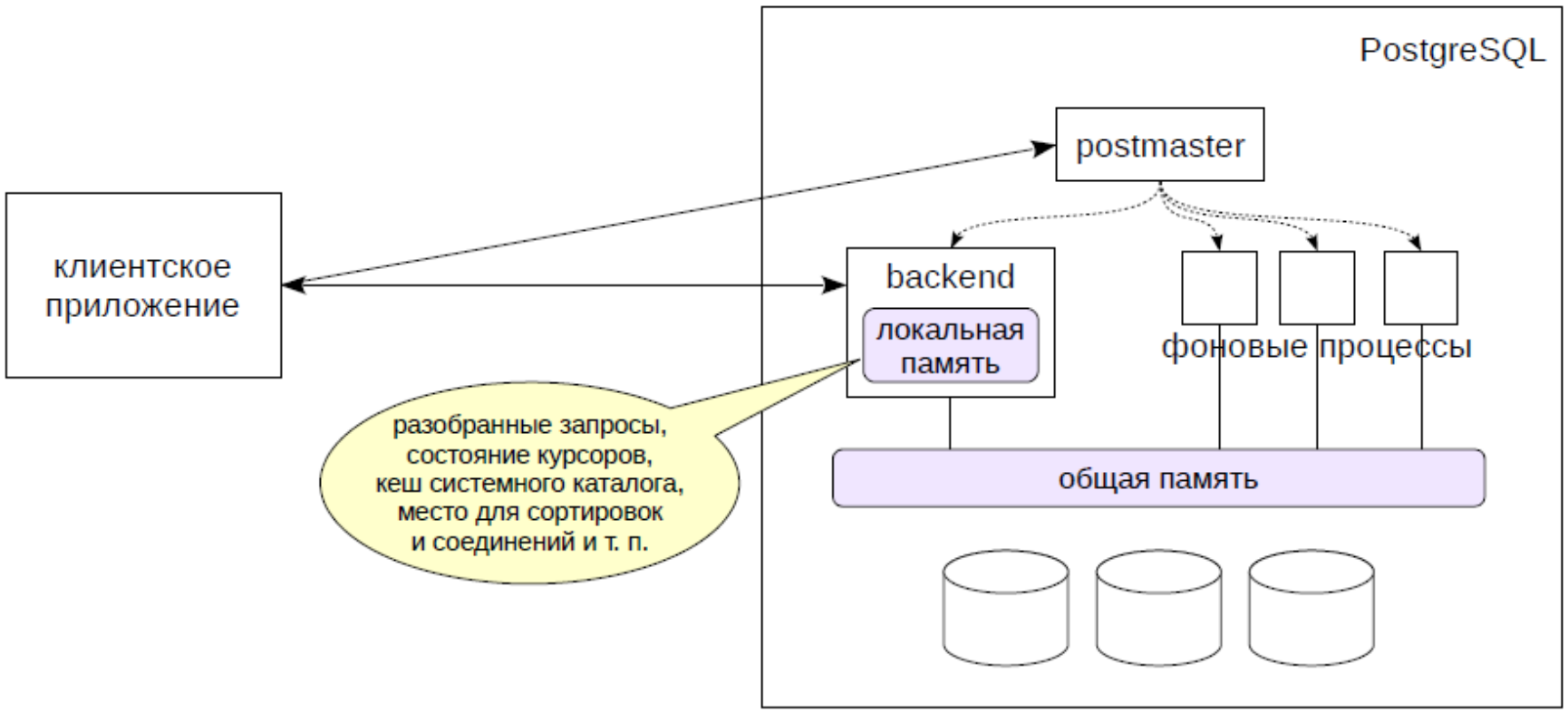
```
CLOSE CURSOR
```

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

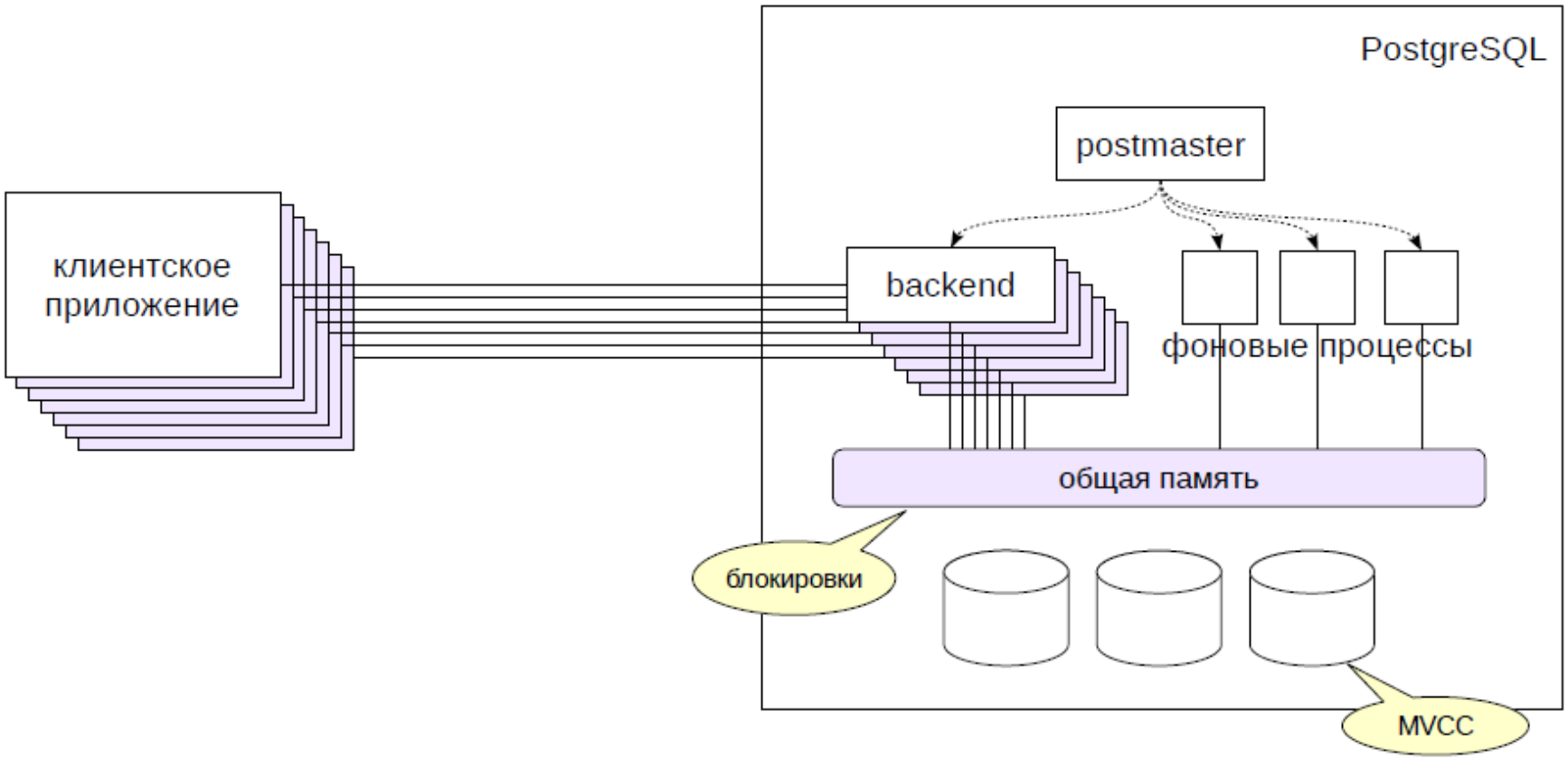
```
=> COMMIT;
```

```
COMMIT
```

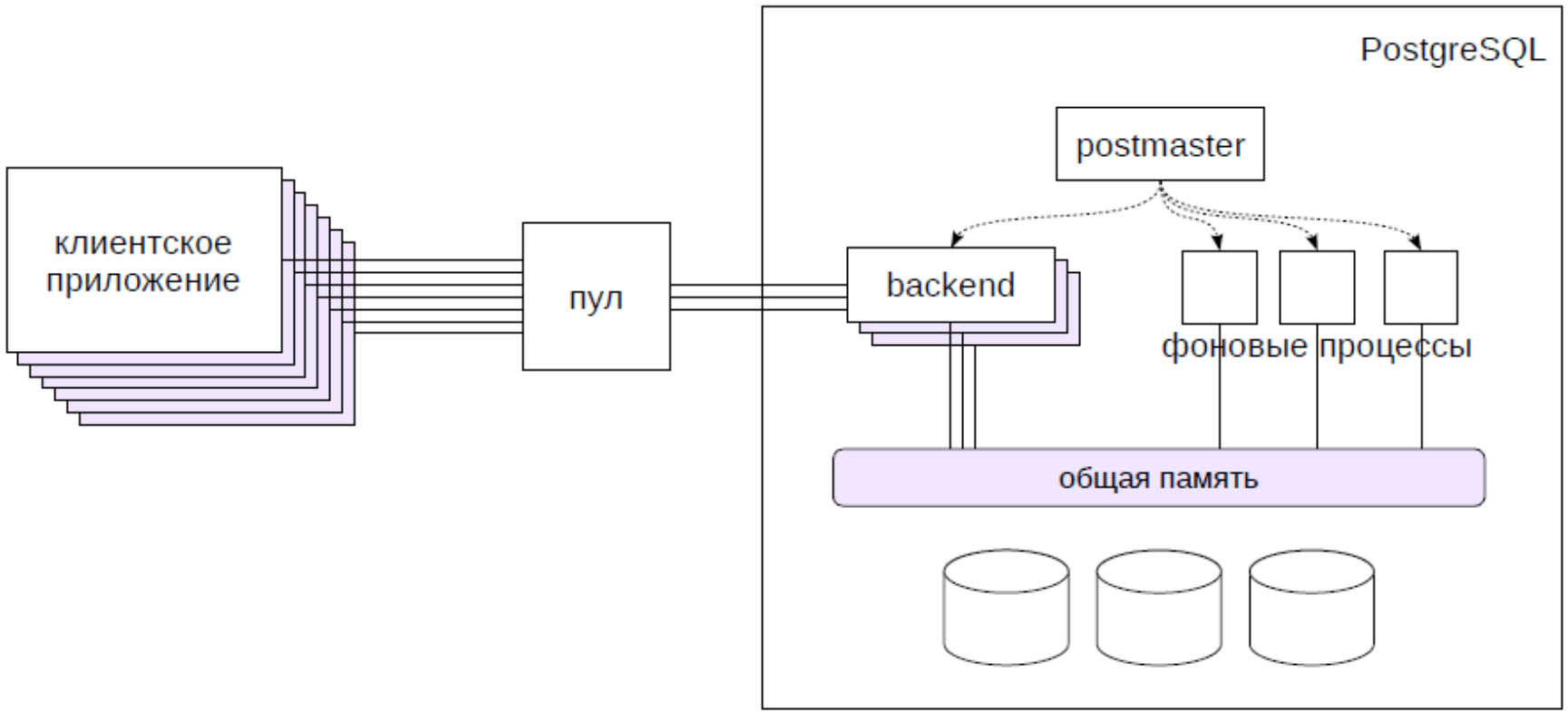
Процессы и память



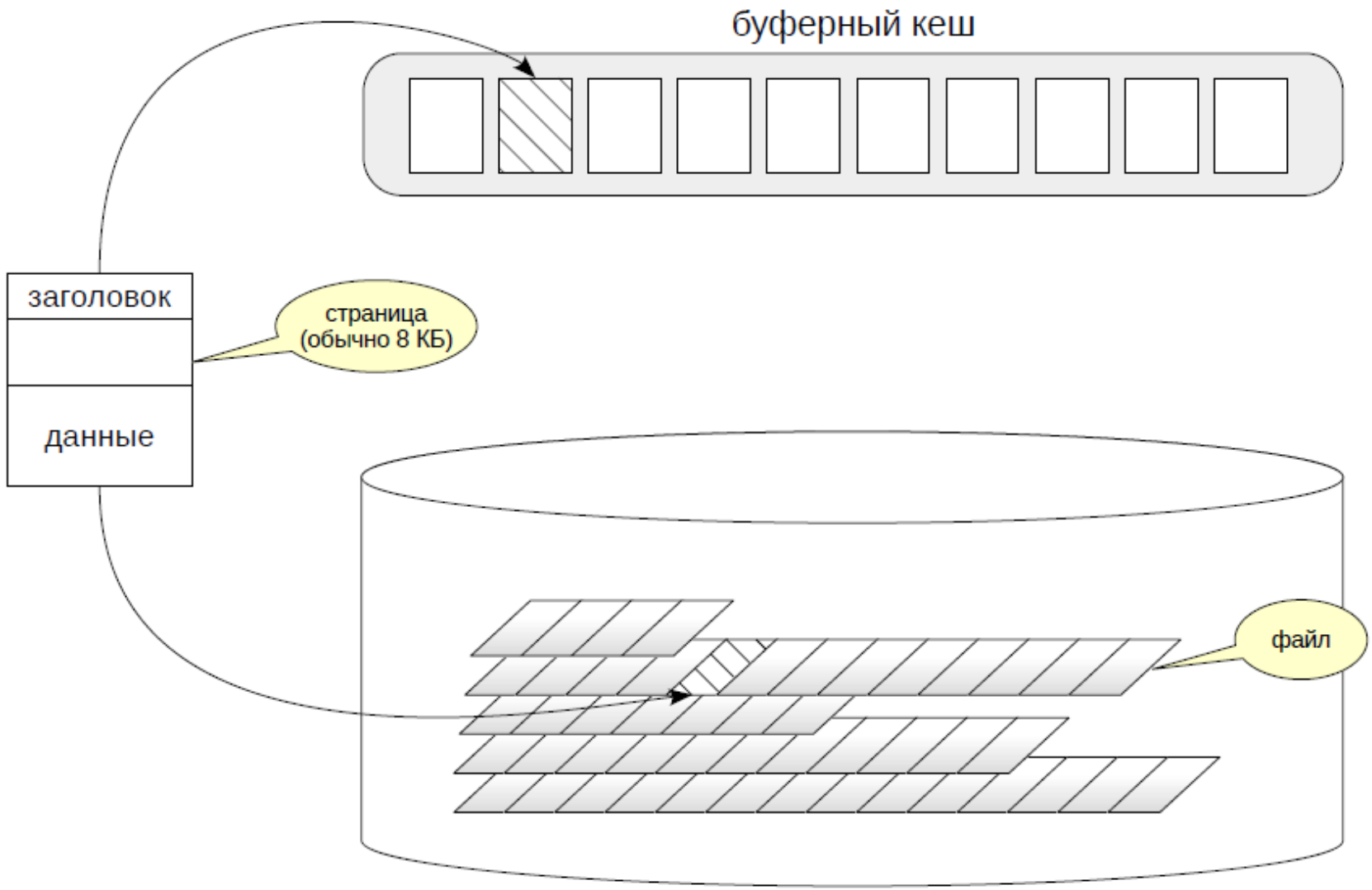
Много клиентов



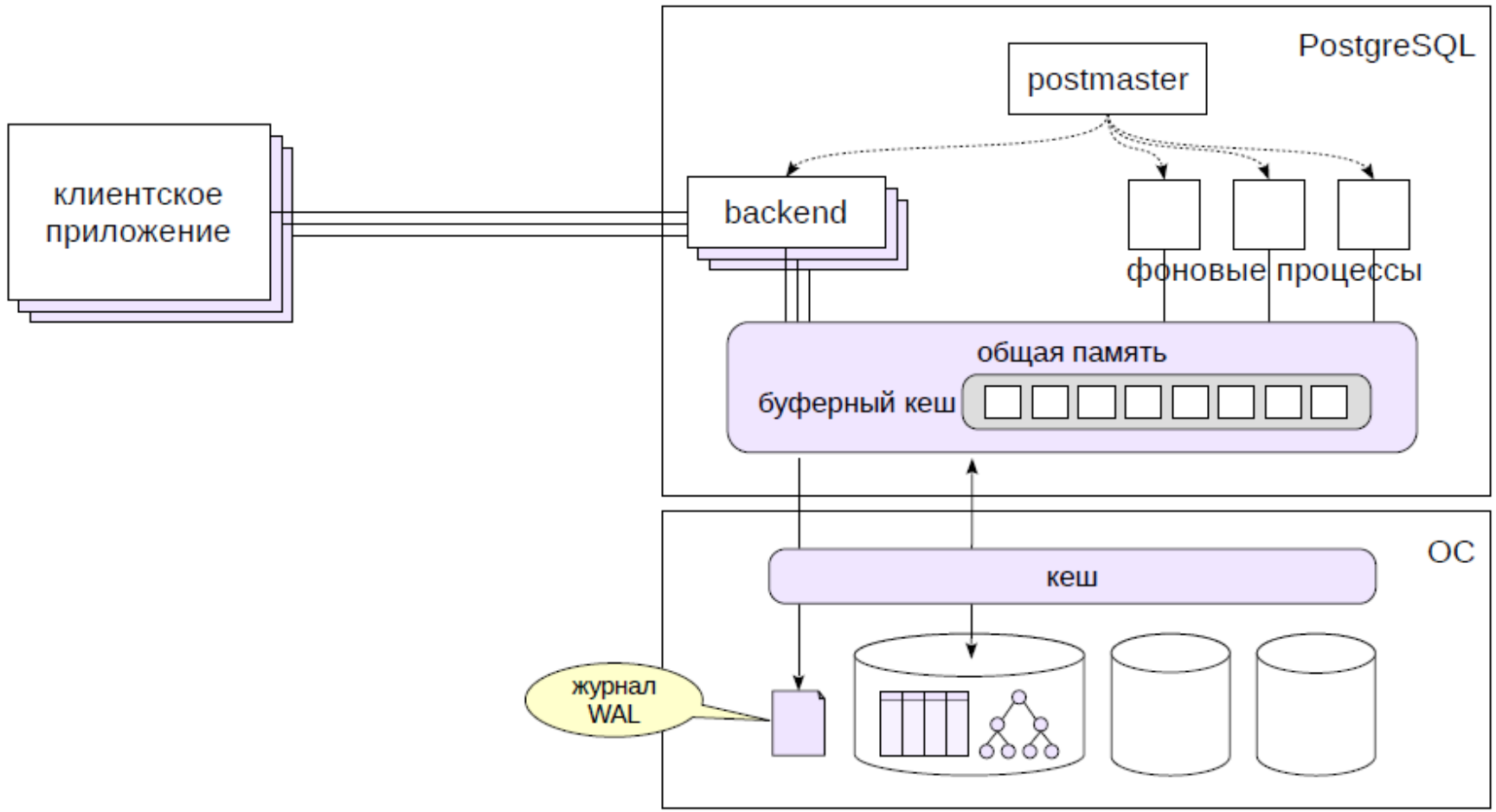
Пул соединений



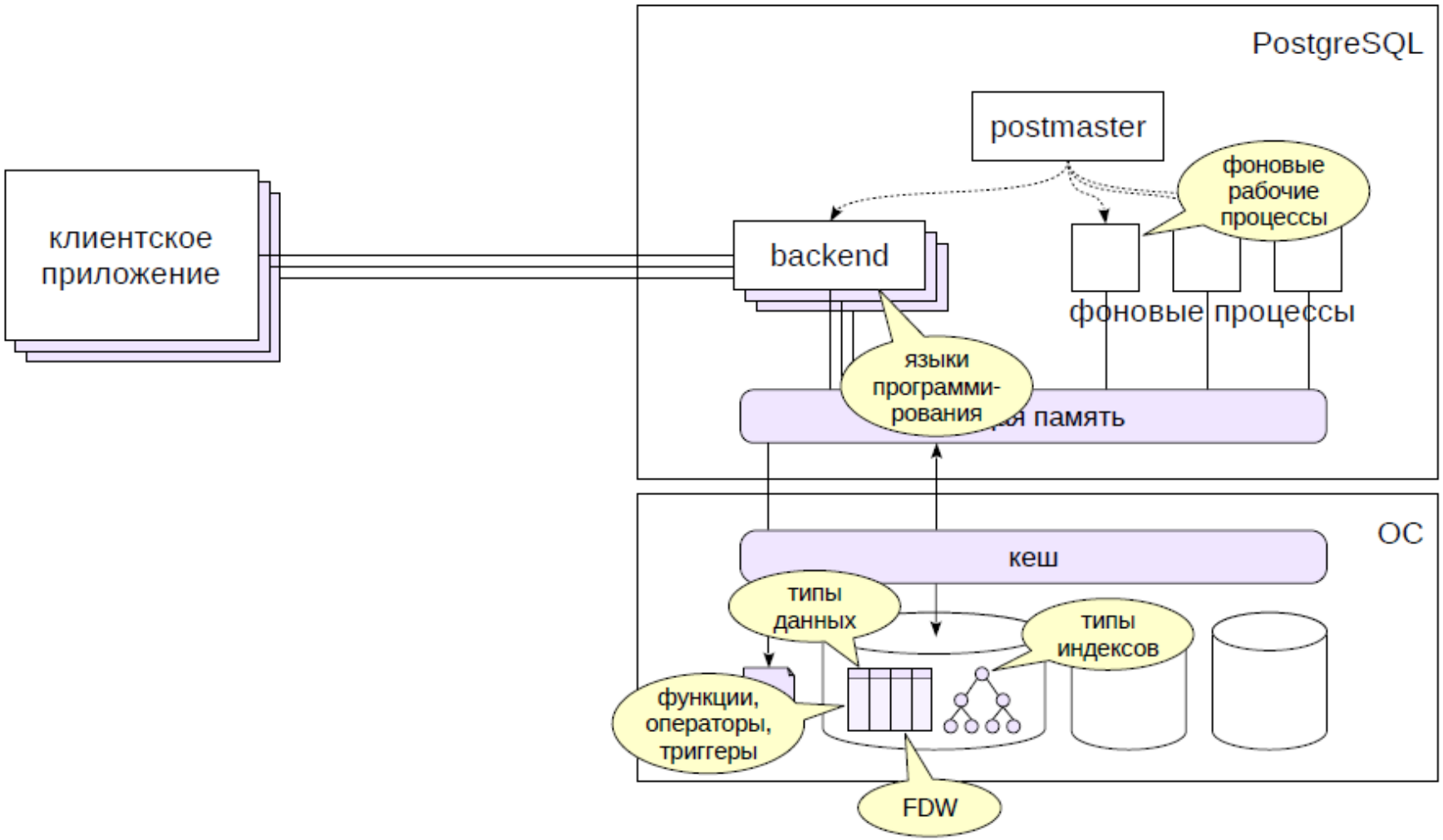
Хранение данных



Хранение данных



Расширяемость



Итоги

Postgres PROFESSIONAL

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)

Архитектура

Изоляция и многоверсионность



Темы

Многоверсионность

Снимок данных

Уровни изоляции

Очистка и ее горизонт

Блокировки

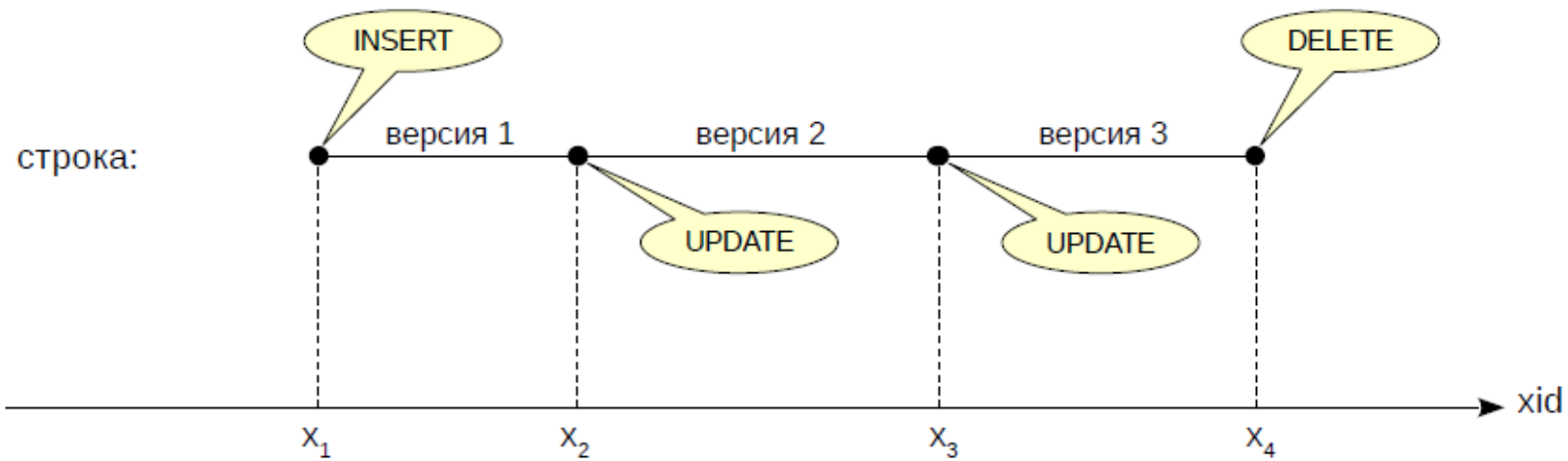
Статус транзакций

Многоверсионность

Наличие нескольких версий одной и той же строки

версии различаются временем действия

время = номер транзакции (номера выдаются по возрастанию)



Снимок данных **ВАЖНО!**

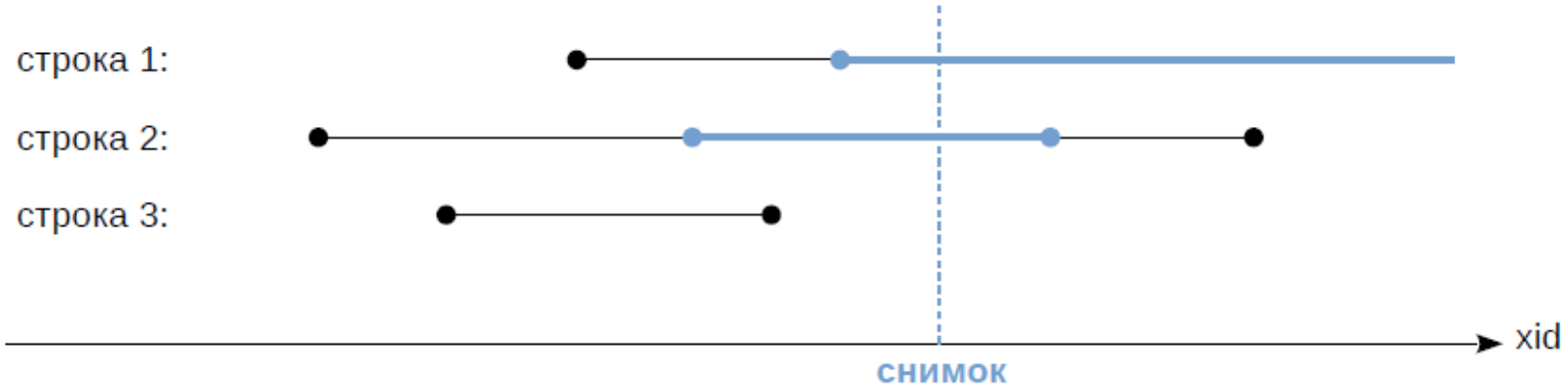
Снимок — это не физическая копия данных, а всего несколько чисел:

Согласованный срез на определенный момент времени

номер транзакции — определяет момент времени

список активных транзакций — чтобы не смотреть на еще не зафиксированные изменения

Список нужен для того, чтобы исключить из снимка изменения тех транзакций, которые начались до создания снимка, но еще не были зафиксированы.

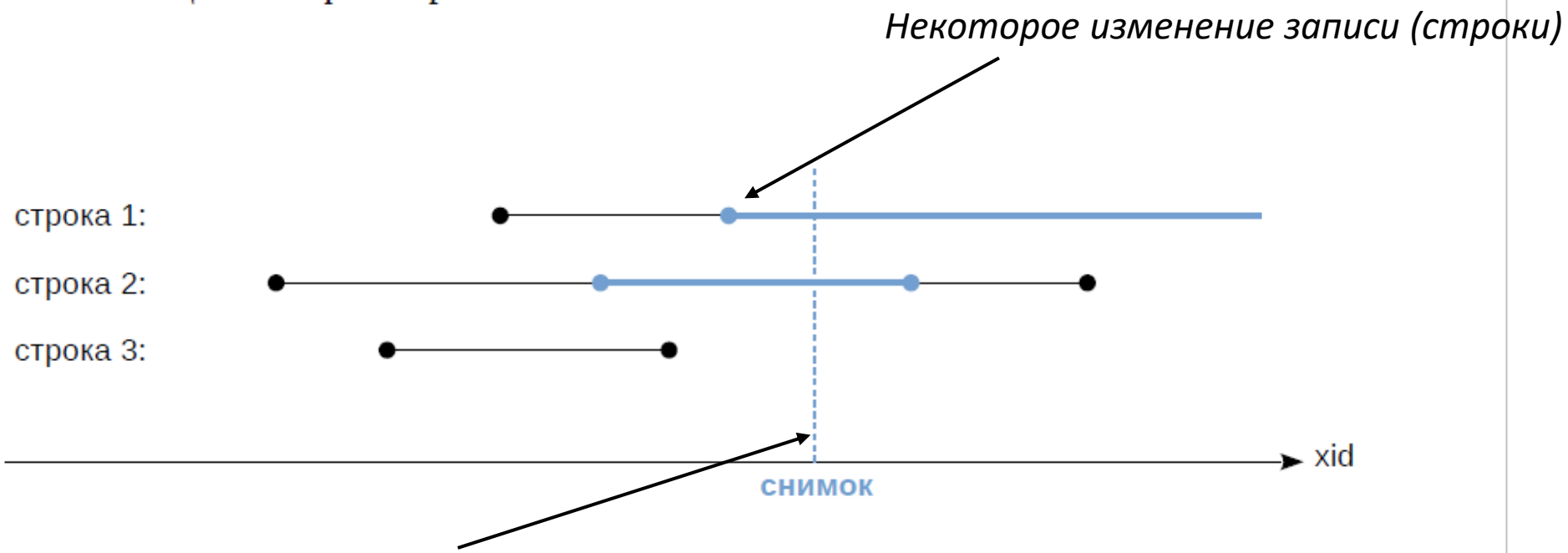


Снимок данных

Согласованный срез на определенный момент времени

номер транзакции — определяет момент времени

список активных транзакций — чтобы не смотреть на еще не зафиксированные изменения

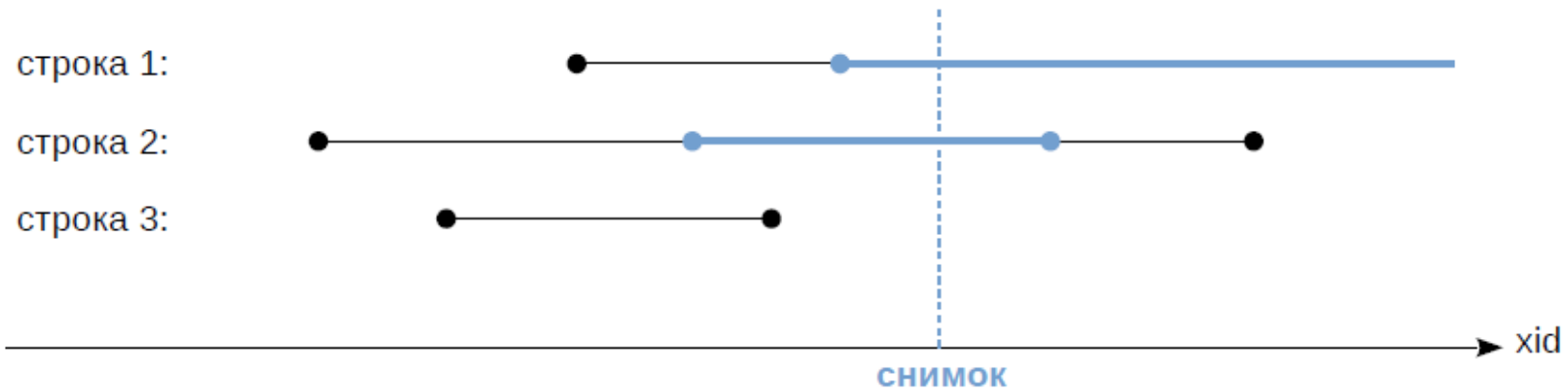


Снимок данных

Согласованный срез на определенный момент времени

номер транзакции — определяет момент времени

список активных транзакций — чтобы не смотреть на еще не зафиксированные изменения



Уровни изоляции

Read Uncommitted

- не поддерживается PostgreSQL: работает как Read Committed

Read Committed — *используется по умолчанию*

- снимок строится на момент начала оператора
- одинаковые запросы могут каждый раз получать разные данные

Repeatable Read

- снимок строится на момент начала первого оператора транзакции
- транзакция может завершиться ошибкой сериализации

Serializable

- полная изоляция, но дополнительные накладные расходы
- транзакция может завершиться ошибкой сериализации

Уровень изоляции	*«Грязное» чтение (dirty read)*	*Неповторяемое чтение (nonrepeatable read)*	*Фантомное чтение (phantom read)*	*Аномалия сериализации (serialization anomaly)*
READ UNCOMMITTED (Чтение Незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
READ COMMITTED (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
REPEATABLE READ (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
SERIALIZABLE (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

Уровень изоляции	*«Грязное» чтение (dirty read)*	*Неповторяемое чтение (nonrepeatable read)*	*Фантомное чтение (phantom read)*	*Аномалия сериализации (serialization anomaly)*
READ UNCOMMITTED (Чтение Незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
READ COMMITTED (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
REPEATABLE READ (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
SERIALIZABLE (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

Снимок на момент начала ОПЕРАТОРА

Снимок на момент начала ТРАНЗАКЦИИ

Видимость версий строк

Как убедиться в том, что одна и та же строка может существовать в нескольких версиях?

Создадим таблицу:

```
=> CREATE TABLE t(s text);
```

```
CREATE TABLE
```

И вставим одну строку. Напомним, что если не начать транзакцию явно командой BEGIN, psql выполняет команду и немедленно фиксирует результат:

```
=> INSERT INTO t VALUES ('Первая версия');
```

```
INSERT 0 1
```

Начнем транзакцию и выведем ее номер:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT pg_current_xact_id();
```

```
 pg_current_xact_id
--------------------
                747
(1 row)
```

Транзакция видит первую (и пока единственную) версию строки:

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Первая версия |  746 |    0
(1 row)
```



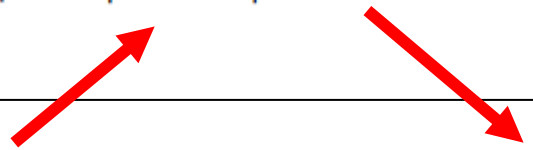
Какая транзакция создала эту версию строки

Какая транзакция изменила эту версию строки (и сделала версию неактуальной/невидимой)

Транзакция видит первую (и пока единственную) версию строки:

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Первая версия |  746 |    0
(1 row)
```



Какая транзакция создала эту версию строки

Начиная с какой транзакции эта версия видна в снимках

Какая транзакция изменила эту версию строки (и сделала версию неактуальной/невидимой)

Начиная с какой транзакции эта версия НЕ видна в снимках

Транзакция видит первую (и пока единственную) версию строки:

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Первая версия |  746 |    0
(1 row)
```

0 – версия актуальна
(никто еще не изменил)

Какая транзакция создала эту версию строки

Начиная с какой транзакции эта версия
видна в снимках

Какая транзакция изменила эту версию строки
(и сделала версию неактуальной/невидимой)

Начиная с какой транзакции эта версия
НЕ видна в снимках

Видимость версий строк

Как убедиться в том, что одна и та же строка может существовать в нескольких версиях?

Создадим таблицу:

```
=> CREATE TABLE t(s text);

CREATE TABLE
```

И вставим одну строку. Напомним, что если не начать транзакцию явно командой BEGIN, psql выполняет команду и немедленно фиксирует результат:

```
=> INSERT INTO t VALUES ('Первая версия');

INSERT 0 1
```

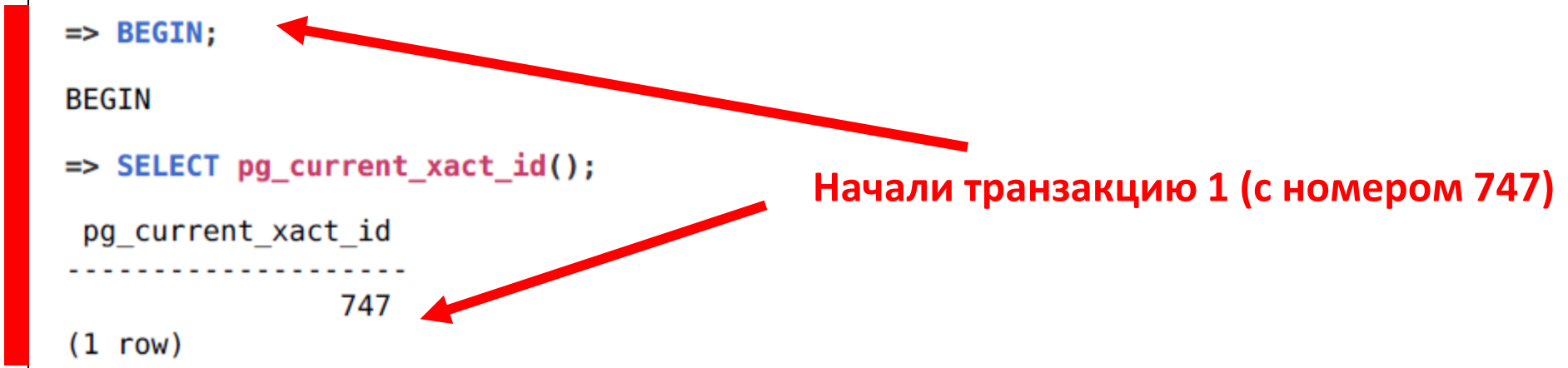
Начнем транзакцию и выведем ее номер:

```
=> BEGIN;

BEGIN

=> SELECT pg_current_xact_id();

 pg_current_xact_id
--------------------
                747
(1 row)
```



Теперь начнем другую транзакцию в другом сеансе:

```
=> BEGIN;

BEGIN

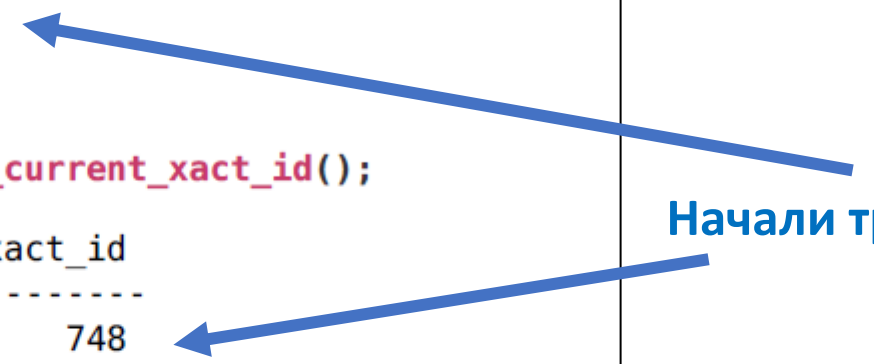
=> SELECT pg_current_xact_id();

 pg_current_xact_id
--------------------
                748
(1 row)
```

Транзакция видит ту же единственную версию:

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Первая версия |  746 |    0
(1 row)
```



Начали транзакцию 2 (с номером 748)

Теперь изменим строку во второй транзакции.

```
=> UPDATE t SET s = 'Вторая версия';

UPDATE 1
```

Вот что получилось:

```
=> SELECT *, xmin, xmax FROM t;
```

```
       s       | xmin | xmax
---------------+------+------
 Вторая версия |  748 |    0
(1 row)
```

Теперь изменим строку во второй транзакции.

```
=> UPDATE t SET s = 'Вторая версия';

UPDATE 1
```

Вот что получилось:

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Вторая версия |  748 |    0
(1 row)
```

А что увидит первая транзакция?

```
=> SELECT *, xmin, xmax FROM t;
```

?

Теперь изменим строку во второй транзакции.

```
=> UPDATE t SET s = 'Вторая версия';

UPDATE 1
```

Вот что получилось:

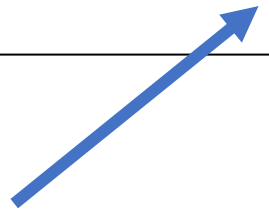
```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Вторая версия |  748 |    0
(1 row)
```

А что увидит первая транзакция?

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Первая версия |  746 |  748
(1 row)
```



Транзакция с номером 748 уже меняла строку

Теперь изменим строку во второй транзакции.

```
=> UPDATE t SET s = 'Вторая версия';

UPDATE 1
```

Вот что получилось:

```
=> SELECT *, xmin, xmax FROM t;

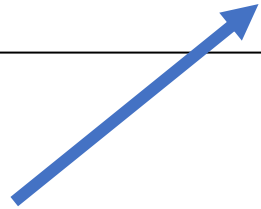
       s       | xmin | xmax
---------------+------+------
 Вторая версия |  748 |    0
(1 row)
```

Вообще говоря, такое «подглядывание» нарушает изоляцию, поэтому поля xmin и xmax скрыты и в реальной работе их использовать не стоит.

А что увидит первая транзакция?

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Первая версия |  746 |  748
(1 row)
```



Транзакция с номером 748 уже меняла строку

Теперь зафиксируем изменения.

```
=> COMMIT;

COMMIT
```

Что теперь увидит первая транзакция?

```
=> SELECT *, xmin, xmax FROM t;

       s       | xmin | xmax
---------------+------+------
 Вторая версия |  748 |    0
(1 row)
```

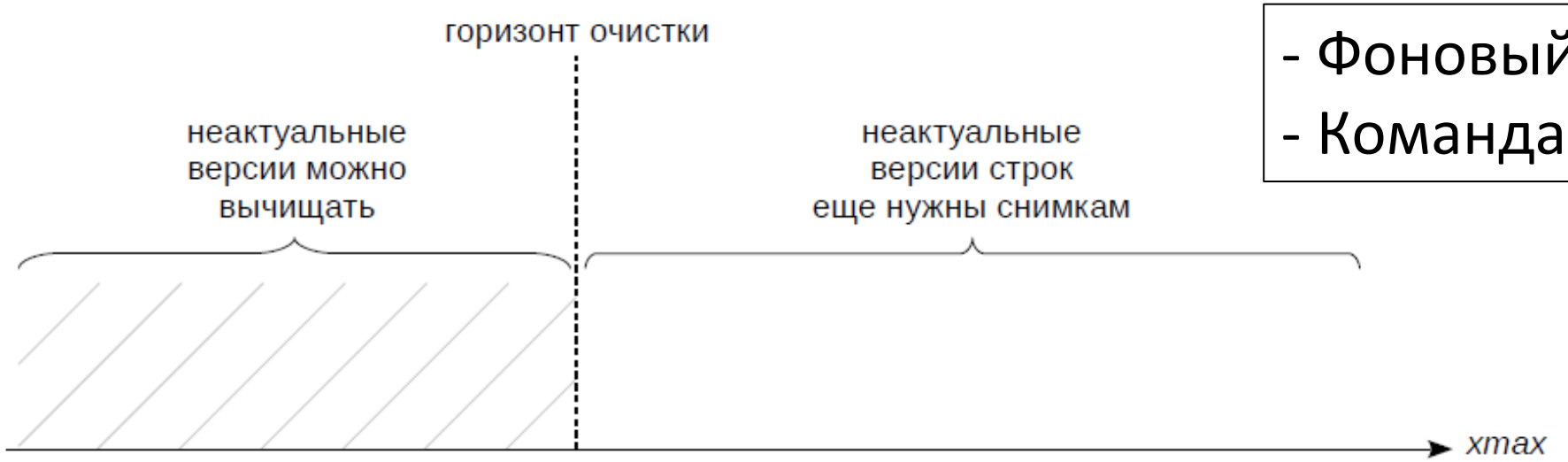
Теперь и первая транзакция видит вторую версию строки.

После фиксации первая версия строки больше не будет видна ни в одной транзакции.

```
=> COMMIT;

COMMIT
```

Очистка и ее горизонт



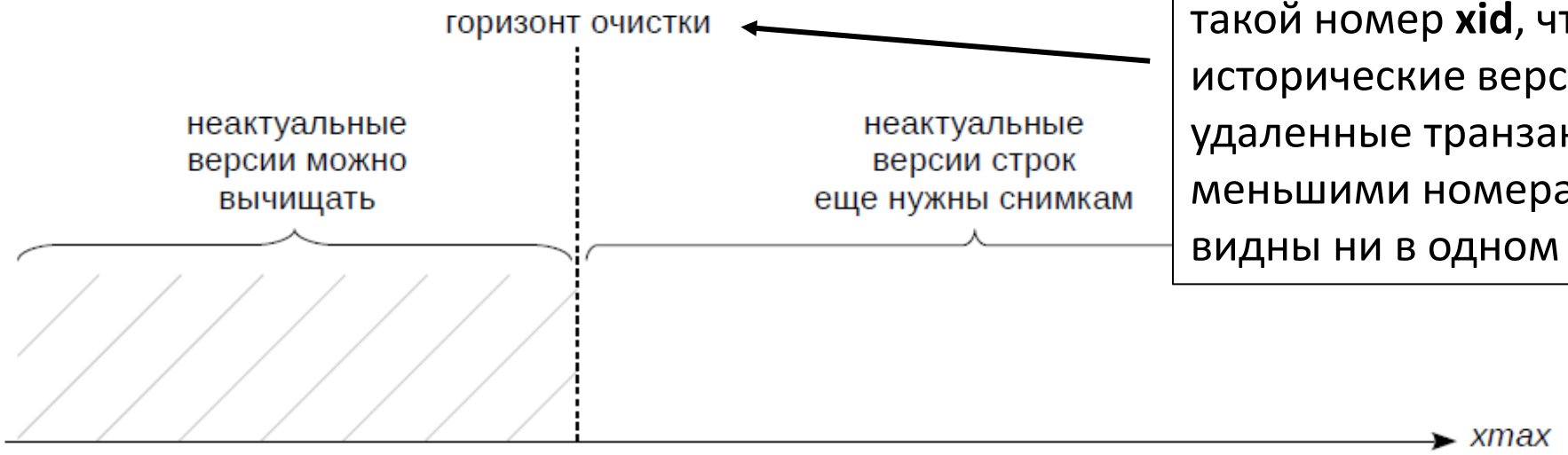
- Фоновый процесс
- Команда VACUUM

Горизонт один на каждую базу данных

Долгие транзакции могут удерживать горизонт

мешая тем самым вычищать неактуальные версии строк

Очистка и ее горизонт



такой номер **xid**, что все исторические версии, удаленные транзакциями с меньшими номерами, уже не видны ни в одном снимке.

Горизонт один на каждую базу данных

Долгие транзакции могут удерживать горизонт

мешая тем самым вычищать неактуальные версии строк

Блокировки

Блокировки строк

- чтение никогда не блокирует строки
- изменение строк блокирует их для изменений, но не для чтений

Блокировки таблиц

- запрещают изменение или удаление таблицы, пока с ней идет работа
- запрещают чтение таблицы при перестроении или перемещении
- и т. п.

Время жизни блокировок

- устанавливаются по мере необходимости или вручную
- снимаются автоматически при завершении транзакции или при откате к точке сохранения

Блокировки

Повторим наш опыт, но теперь пусть обе транзакции попытаются изменить одну и ту же строку.

```
=> BEGIN;

BEGIN

=> UPDATE t SET s = 'Третья версия' RETURNING *;

       s
---------------
 Третья версия
(1 row)

UPDATE 1
```

И во второй транзакции:

```
   => BEGIN;

   BEGIN

   => UPDATE t SET s = 'Четвертая версия' RETURNING *;
```

Вторая транзакция «повисла»: она не может изменить строку, пока первая транзакция не снимет блокировку.

```
=> COMMIT;

COMMIT
```

Теперь вторая транзакция может продолжить выполнение:

```
         s
------------------
 Четвертая версия
(1 row)

UPDATE 1

=> COMMIT;

COMMIT
```

Обе транзакции зафиксировали свои изменения. Первый сеанс снова читает таблицу и видит актуальную строку — это результат, зафиксированный второй транзакцией:

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

Новая транзакция от первого сеанса

```
         s
------------------
 Четвертая версия
(1 row)
```

Статус транзакций

Статус транзакций (clog)

- служебная информация; два бита на транзакцию
- специальные файлы на диске
- буферы в общей памяти

Фиксация

- устанавливается бит «транзакция зафиксирована»

Обрыв

- устанавливается бит «транзакция прервана»
- выполняется так же быстро, как и фиксация (не нужен откат данных)

хранятся в специальных служебных файлах в каталоге PGDATA/pg_xact

Итоги

В файлах данных могут храниться несколько версий каждой строки

Транзакции работают со снимком данных — согласованным срезом на определенный момент времени

Уровни изоляции отличаются временем создания снимка

Мертвые версии строк за горизонтом очистки должны периодически вычищаться

Писатели не блокируют читателей,
читатели не блокируют никого

Архитектура

Буферный кеш и журнал



Темы

Устройство буферного кеша

Алгоритм вытеснения

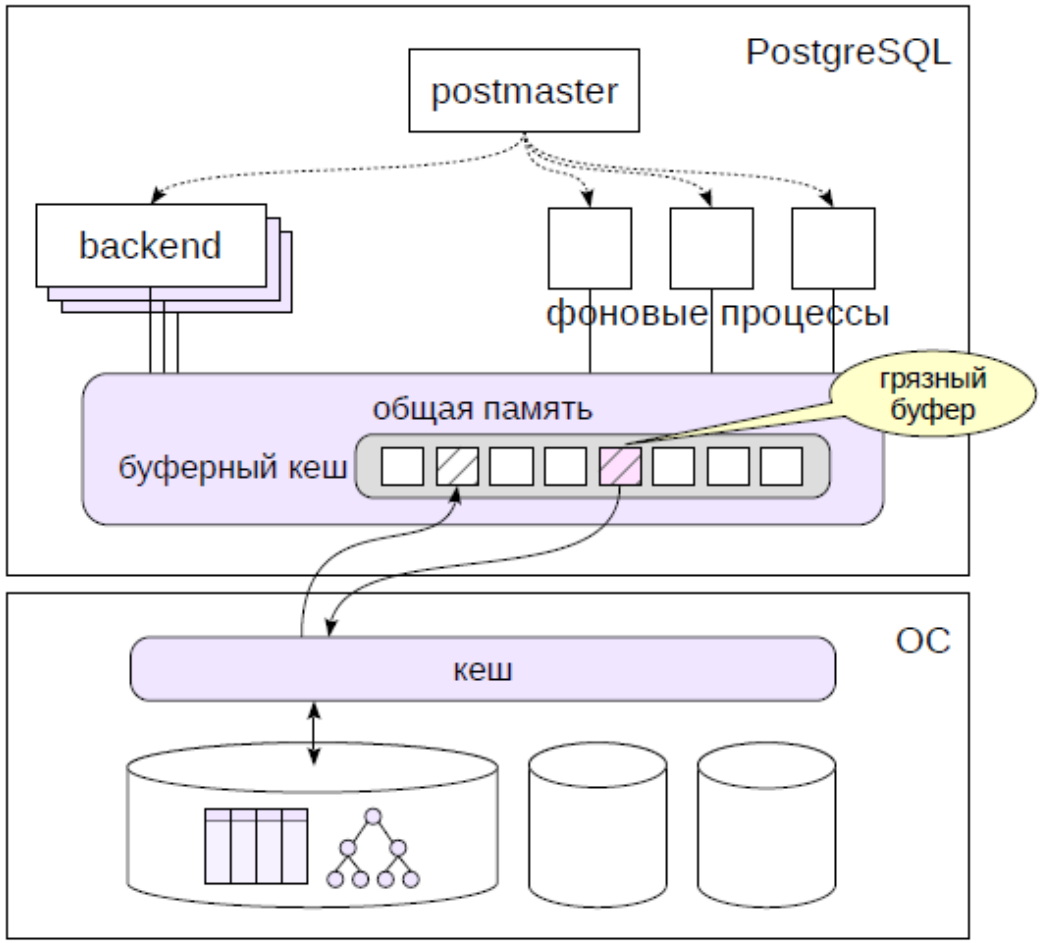
Журнал предзаписи

Контрольная точка

Процессы, связанные с буферным кешем и журналом

Буферный кеш

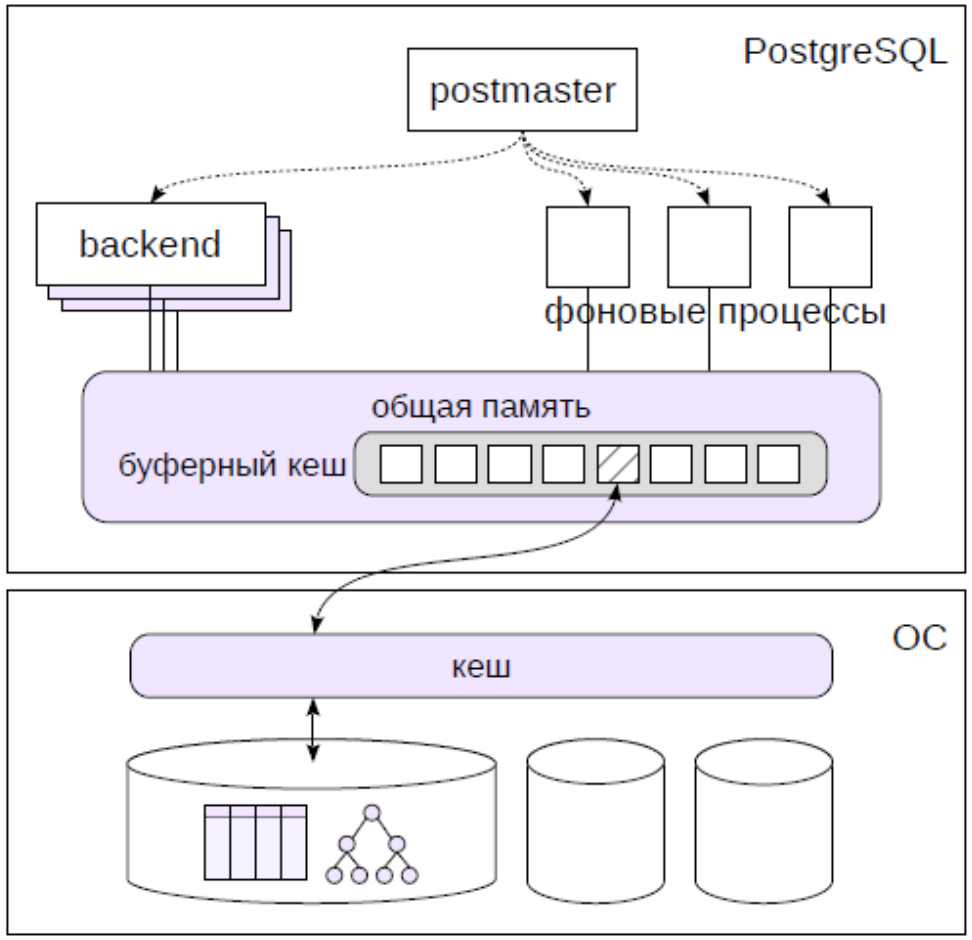
- Массив буферов
  - страница данных (8 Кбайт)
  - доп. информация
- «Грязные» буферы
  - асинхронная запись
- Блокировки в памяти
  - для совместного доступа



Вытеснение

Вытеснение редко используемых страниц

- грязный буфер записывается на диск
- на освободившееся место читается другая страница



Алгоритм вытеснения LRU — Least Recently Used

Влияние буферного кеша на выполнение запросов

Создаем новую БД в кластере и подключаемся к ней (подробнее про базы данных — в модуле «Организация данных»):

```
=> CREATE DATABASE arch_wal_overview;

CREATE DATABASE

=> \c arch_wal_overview

You are now connected to database "arch_wal_overview" as user "student".
```

Создадим таблицу:

```
=> CREATE TABLE t(n integer);

CREATE TABLE
```

Заполним таблицу некоторым количеством строк:

```
=> INSERT INTO t SELECT id FROM generate_series(1,100_000) AS id;

INSERT 0 100000
```

Размер буферного кеша показывает параметр shared_buffers:

```
=> SHOW shared_buffers;

 shared_buffers
----------------
 128MB
(1 row)
```

Теперь перезапустим сервер, чтобы содержимое буферного кеша сбросилось.

```
=> \q

student$ sudo pg_ctlcluster 16 main restart

student$ psql arch_wal_overview
```

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM t;

                  QUERY PLAN
-----------------------------------------------
 Seq Scan on t (actual rows=100000 loops=1)
   Buffers: shared read=443
 Planning:
   Buffers: shared hit=12 read=8 dirtied=1
 Planning Time: 0.146 ms
 Execution Time: 6.177 ms
(6 rows)
```

«Buffers: shared» – использование буферного кеша.

- **read** — количество буферов, в которые пришлось прочитать страницы с диска.

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM t;

                  QUERY PLAN
-----------------------------------------------
 Seq Scan on t (actual rows=100000 loops=1)
   Buffers: shared read=443
 Planning:
   Buffers: shared hit=12 read=8 dirtied=1
 Planning Time: 0.146 ms
 Execution Time: 6.177 ms
(6 rows)
```

«Buffers: shared» – использование буферного кеша.

- **read** — количество буферов, в которые пришлось прочитать страницы с диска.

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM t;

                  QUERY PLAN
-----------------------------------------------
 Seq Scan on t (actual rows=100000 loops=1)
   Buffers: shared hit=443
 Planning Time: 0.028 ms
 Execution Time: 4.306 ms
(4 rows)
```

- **hit** — количество буферов, в которых нашлись нужные для запроса страницы..

во второй раз уменьшилось и время выполнения запроса, и время его планирования (потому что таблицы системного каталога тоже кешируются).

Журнал предзаписи (WAL)

Файлы журнала располагаются в каталоге PGDATA/pg_wal.

Проблема: при сбое теряются данные из оперативной памяти, не записанные на диск

Журнал

- поток информации о выполняемых действиях,
  позволяющий повторно выполнить потерянные при сбое операции
- запись попадает на диск раньше, чем измененные данные

Журнал защищает

- страницы таблиц, индексов и других объектов
- статус транзакций (clog)

Журнал не защищает

- временные и нежурналируемые таблицы

Журнал предзаписи

Логически журнал можно представить в виде непрерывного потока записей. Каждая запись имеет номер, называемый LSN (Log Sequence Number). Это 64-разрядное число — смещение записи в байтах относительно начала журнала.

Текущую позицию показывает функция pg_current_wal_lsn:

```
=> SELECT pg_current_wal_lsn();

 pg_current_wal_lsn
--------------------
 0/23BCD28
(1 row)
```



Log Sequence Number

Позиция записывается как два 32-разрядных числа через косую черту. Запомним это значение.

Выполним теперь какие-нибудь операции и посмотрим, как изменилась позиция.

```
=> UPDATE t SET n = 100_001 WHERE n = 1;

UPDATE 1

=> SELECT pg_current_wal_lsn();

 pg_current_wal_lsn
--------------------
 0/23BFD40
(1 row)
```

Интересны не абсолютные числа, а их разница, которая показывает размер сгенерированных журнальных записей в байтах:

```
=> SELECT '0/23BFD40'::pg_lsn - '0/23BCD28'::pg_lsn AS bytes;

 bytes
-------
 12312
(1 row)
```

Физически журнал хранится в файлах в отдельном каталоге (PGDATA/pg_wal). По умолчанию файлы имеют размер 16 Мбайт, задать другой размер можно при инициализации кластера баз данных.

На файлы можно взглянуть не только средствами файловой системы, но и с помощью функции:

```
=> SELECT * FROM pg_ls_waldir() ORDER BY name LIMIT 10;

            name            |   size   |      modification
----------------------------+----------+------------------------
 000000010000000000000002 | 16777216 | 2025-06-24 04:30:27+03
 000000010000000000000003 | 16777216 | 2025-06-24 04:30:18+03
(2 rows)
```

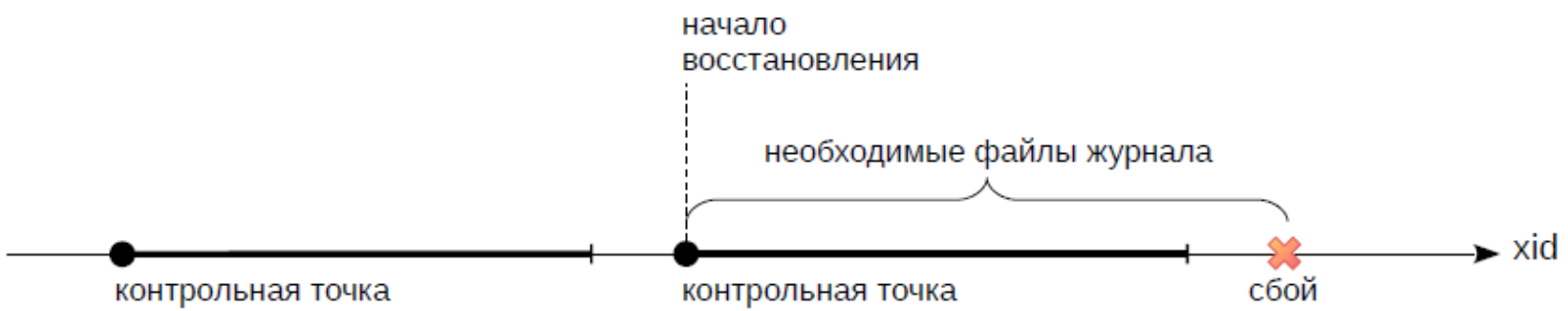
Контрольная точка

Периодический сброс всех грязных буферов на диск

- гарантирует попадание на диск всех изменений до контрольной точки
- ограничивает размер журнала, необходимого для восстановления

Восстановление при сбое

- начинается с последней контрольной точки
- последовательно проигрываются записи, если изменений нет на диске



Восстановление при помощи журнала

Измененные табличные страницы находятся в буферном кеше, но еще не записаны на диск. При обычной остановке сервер выполняет контрольную точку, чтобы записать все грязные страницы на диск, но мы сымитируем сбой системы, послав сигнал процессу postmaster.

```
student$ sudo head -n 1 /var/lib/postgresql/16/main/postmaster.pid
```

```
7700
```

```
student$ sudo kill -QUIT 7700
```

При старте произойдет восстановление согласованности данных с помощью журнала. Проверим:

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

```
student$ psql arch_wal_overview
```

```
=> SELECT min(n), max(n) FROM t;
```

```
 min |  max
-----+--------
   2 | 100001
(1 row)
```

Все изменения были восстановлены.

PostgreSQL автоматически удаляет журнальные файлы, не требующиеся для восстановления, после выполнения контрольной точки.

Производительность

Синхронный режим

запись при фиксации

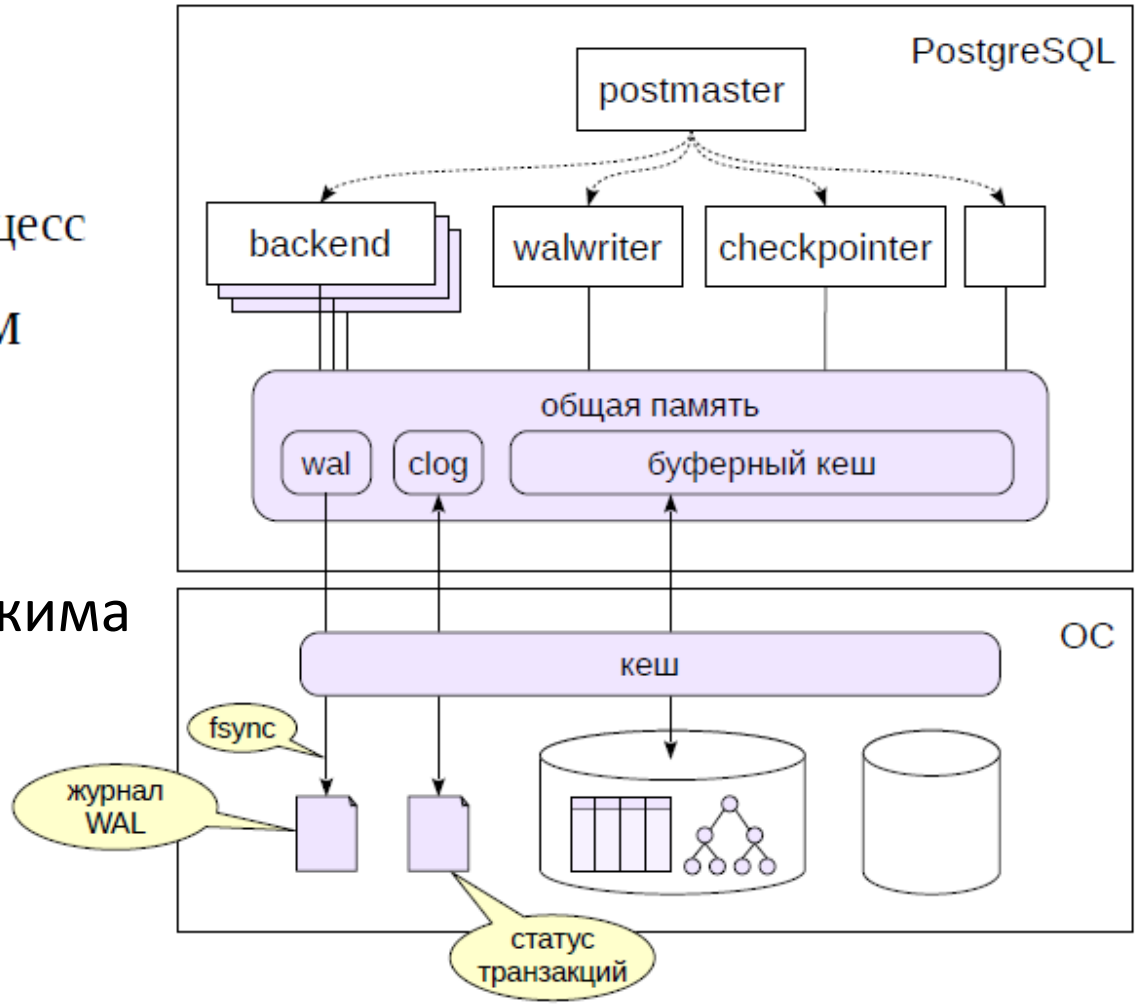
обслуживающий процесс

Асинхронный режим

фоновая запись

walwriter

На самом деле оба режима работают совместно



Основные процессы

Запись журнала

Контрольная точка

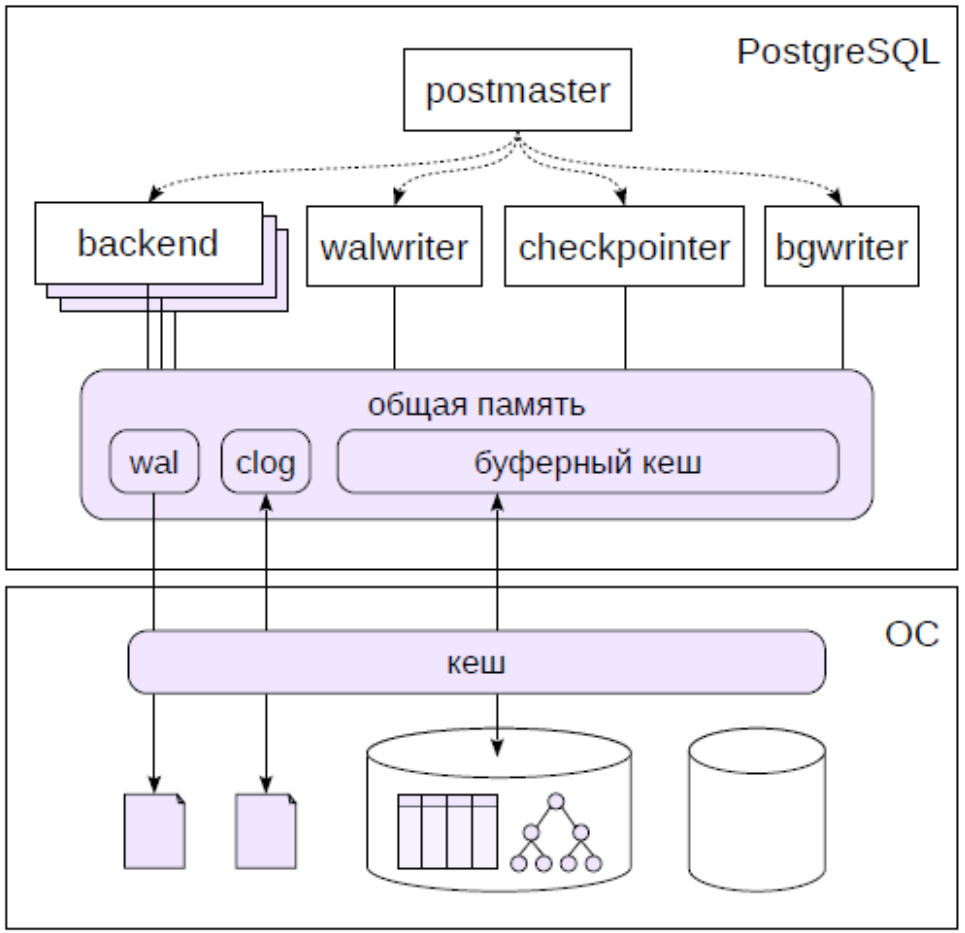
- сброс всех грязных буферов

Фоновая запись

- сброс части грязных буферов

Обслуживающие процессы

- сброс вытесняемого грязного буфера



Уровни журнала

Minimal

- гарантия восстановления после сбоя

Replica *(по умолчанию)*

- резервное копирование
- репликация: передача и проигрывание журнала на другом сервере

Logical

- логическая репликация: информация о добавлении, изменении и удалении табличных строк

Итоги

Буферный кеш существенно ускоряет работу,
уменьшая число дисковых операций

Надежность обеспечивается журналированием

Размер журнала ограничен благодаря контрольным точкам

Журнал удобен и используется во многих случаях

- для восстановления после сбоя
- при резервном копировании
- для репликации между серверами