2

2

# Балансировка нагрузки

# Клиент-серверный протокол

- Сессия обслуживается отдельным серверным процессом
  - процесс создаётся до аутентификации
- Клиент и серверный процесс обмениваются сообщениями по TCP/IP или Unix-сокетам
  - Для аутентификации может использоваться одно или несколько сетевых сообщений
  - Сообщение перед передачей помещается в буфер
- Простой протокол - передача текста команды
- Расширенный протокол - подготовка команды, привязка параметров, выполнение
- Готовая к выполнению, подготовленная команда называется порталом
  - Открытый курсор - частный случай портала



## Клиент-серверный протокол

Для взаимодействия клиентов с экземпляром PostgreSQL используется протокол, основанный на обмене сообщениями. Протокол использует сокеты TCP/IP или Unix-сокеты. Для серверной части, работающей по протоколу, в IANA зарегистрирован порт 5432, но может использоваться любой другой непривилегированный порт (не ниже 1024). В 18 версии PostgreSQL используется версия протокола 3.2, которая совместима с версией 3.0, использующейся, начиная с PostgreSQL 7.4. Версию 3.1 никогда не использовали, её не стали использовать для обхода ошибки в pgbouncer.

Для каждой клиентской сессии создаётся отдельный серверный (синоним обслуживающий, backend) процесс. Серверный процесс создаётся, как только поступает запрос на соединение, до аутентификации.

Для аутентификации может использоваться одно или несколько сетевых сообщений, в зависимости от способа аутентификации.

Данные по клиент-серверному протоколу могут передаваться в текстовом и бинарном (двоичном, binary) форматах. Клиент указывает формат для каждого передаваемого значения переменной привязки и для каждого столбца результат запроса.

Сообщение перед передачей, обычно, помещается в буфер как клиентом, так и серверным процессом, чтобы не допустить передачи неполного сообщения. Буфера выделяется в локальной памяти процесса.

Запросы могут передаваться по простому (simple) и расширенному (extended-query) протоколу. В расширенном протоколе используются подготовленные запросы (prepared statements). Подготовленный запрос - это результат синтаксического *разбора* (*parse*) и семантического анализа текста запроса. В запросе могут быть параметры и они могут передаваться отдельно. Без значений параметров запрос не может выполняться. Передача параметров для запроса (переменных привязки) называется привязкой (bind). После привязки всех параметров (готовый к выполнению) подготовленный запрос называется порталом (portal). Безымянный портал уничтожается в конце транзакции или при выполнении следующей команды Bind, в которой в качестве целевого выбирается безымянный портал. Посылка сообщения Query тоже уничтожает безымянный портал.

Открытый курсор (использует команду `SELECT`) - частный случай портала.

https://docs.tantorlabs.ru/tdb/ru/18_3/se/protocol-overview.html

# Типы сообщений протокола

- Start-up - сообщение о создании соединения
- Simple Query:

```
-> Query("select id, name from tab")
<- RowDescription( num_fields=2,
   "id", INT4OID, text/~~binary~~
   "name", TEXTOID, text/~~binary~~)
<- DataRow("111","alice")
<- DataRow("222","bob")
<- ReadyForQuery('I') где I -idle, T - in transaction, E - error
```

- Extended Query:
- COPY
- Logical replication
- Physical replication
- Cancel Request (Ctrl+C)

```
-> Parse("select id, name from tab where id = $1")
-> Bind(params=[111], column_formats=[binary, text])
-> Describe('')
-> Execute()
<- RowDescription( num_fields=2,
   "id", INT4OID, ~~text~~/binary
   "name", TEXTOID, text/~~binary~~)
<- DataRow(111,"alice")
<- ReadyForQuery('I')
```



## Типы сообщений протокола

По протоколу могут передаваться сообщения:

1) **Start-up** - передача клиентом стартового сообщения для создания соединения. Передаётся версия протокола, имя пользователя (`user`), имя базы данных (`dbname`). Для физической репликации имя базы данных не передаётся, передаётся параметр `replication` со значением 1:

```
psql "replication=1" -c "IDENTIFY_SYSTEM;"
```

Для логической репликации передаётся и имя базы и параметр `replication` со значением `database`:

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

Сервер отвечает сообщением об аутентификации: `ErrorResponse` - отказ в доступе для правила `reject` в `pg_hba.conf`, `AuthenticationOk` для правила `trust` или другим сообщением о способе аутентификации.

Для SSL и GSS (kerberos), сначала посылается запрос SSLRequest или GSSENCRequest, сервер отвечает "да" или "нет". Если "да", то проводится согласование шифрованного соединения и только потом посылается сообщение Start-up.

После аутентификации сервер передаёт сообщение `BackendKeyData(pid, secret)` для того, чтобы клиент мог прерывать выполнение запросов.

2) **Simple Query**. Команда, передаётся в виде строки.

3) **Extended Query**. Текстовый или двоичный формат выбирает клиент при привязке значения параметра (Bind), вне зависимости от команды SQL. Тип параметра можно не указывать, задав для него значение ноль (неопределённый тип) или сделав массив с OID типов параметров короче, чем число параметров (`$n`) в запросе. В примере на слайде массив в `Bind` не передаётся (https://docs.tantorlabs.ru/tdb/ru/18_3/se/protocol-message-formats.html ). В `column_formats` можно указать ожидаемый формат столбцов результата. В сообщении `Describe(строка)` передаётся имя существующего портала, пустая строка обозначает безымянный портал. Расширенный протокол, обычно, используют для многократного выполнения запроса с разными значениями параметров привязки: Bind(), Execute(), Bind(), Execute()...

4) **COPY**

5, 6) **Logical replication, Physical replication** - сообщения потоковой репликации

7) **Cancel Request** - прерывание выполняющейся команды (**Ctrl+C**). Клиент передаёт сообщение `CancelRequest(pid, secret)`.

# Типы сообщений для COPY

- COPY позволяет осуществлять высокоскоростное копирование данных на сервер или с сервера
- команда `COPY` работает со всеми столбцами одинаково: либо в текстовом, либо в бинарном виде (`COPY ... WITH BINARY`)
- Двустороннее копирование запускается, когда клиент, подсоединённый к walsender выполняет команду `START_REPLICATION`

```
 -> Prepare("COPY TO stdout
(select
generate_series(100))")
 -> Bind()
 -> Execute()
 <- CopyOutResponse()
 <- CopyData("набор байт")
 <- CopyData("набор байт")
 <- CopyDone()
 -> Sync
```

```
 -> Prepare("COPY FROM stdin")
 -> Bind()
 -> Execute()
 <- CopyInResponse()
 -> CopyData("набор байт")
 -> CopyData("набор байт")
 -> CopyDone()
 -> Sync
```



## Типы сообщений для COPY

Команда COPY позволяет осуществлять высокоскоростное копирование данных на сервер или с сервера. Команда копирования переключает соединение в режим ("подпротокол"), который действует до завершения передачи данных.

Выполняя `COPY FROM stdin` сервер передаёт клиенту сообщение `CopyInResponse()`. Клиент передаёт ноль или более сообщений `CopyData` с данными. **Данные не обязательно должны совпадать с границами строк, это просто набор байт**. Клиент может завершить приём данных, передав сообщение `CopyDone` (успешно) или `CopyFail` (команда `COPY` завершится с ошибкой). Сервер вернётся в обычный режим обработки, в котором он находился до выполнения команды `COPY` (это может быть простой или расширенный протокол запросов). Затем сервер отправит сообщение `CommandComplete` или `ErrorResponse`.

Если сервер передал `ErrorResponse`, то он ожидает от клиента сообщения Sync, после которого выдаст `ReadyForQuery` и сможет принимать следующие запросы. Пока идёт передача данных (до получения `CopyDone` или `CopyFail`) сервер игнорирует сообщения `Sync` и `Flush`.

Есть режим двустороннего копирования, обеспечивающий высокоскоростную передачу данных с сервера и на сервер. Двустороннее копирование запускается, когда клиент, подсоединённый к walsender выполняет команду `START_REPLICATION`. В этом режиме используются сообщения `CopyData`, `CopyDone`, `CopyBothResponse`. Эти три сообщения содержат поля, из которых клиент может узнать число столбцов в строке и код формата. Код формата может указываться для каждого столбца отдельно, но в текущей реализации, команда `COPY` работает со всеми столбцами одинаково: либо в текстовом, либо в бинарном виде (`COPY ... WITH BINARY`).

# Типы сообщений для Cancel request

- Выполняющиеся команды можно прервать
- При создании сессии, серверный процесс передаёт клиенту сообщение `BackendKeyData(pid, secret)`
- Чтобы прервать выполнение запроса, клиент должен установить новое соединение к серверу и отправить ему сообщение `CancelRequest`, вместо `StartupMessage`, передаваемого при создании сессии
  - аутентификация клиента не производится, для выполнения запроса достаточно правильных `pid+secret`
  - сервер закрывает соединение и не возвращает результат
  - результат `CancelRequest` будет виден в исходной сессии - запрос выдаст ошибку



## Типы сообщений для Cancel request

Выполняющиеся команды можно прервать. Возможная причина: если команда выполняется неожиданно долго. Например, в psql дана команда `drop table` и команда ожидает получения блокировки. Клиент хочет отказаться от её выполнения. Для этого в psql можно набрать комбинацию клавиш `ctrl+c`.

При создании сессии, серверный процесс передаёт клиенту сообщение `BackendKeyData(pid, secret)`. До 18 версии `secret` был фиксированной длины 4 байта, начиная **с 18 версии** (протокол версии 3.2) поле `secret` имеет переменную длину до 256 байт. Клиент сохраняет эти данные, чтобы впоследствии передавать сообщения `CancelRequest`, которые прерывают выполнение команд в сессии клиента.

Передать команду серверному процессу по соединению нельзя, так как серверный процесс однопоточный, занимается выполнением команды и не реагирует на сообщения. Поэтому, для прерывания команд предусмотрено в клиент-серверном протоколе предусмотрен отдельный тип сообщения, которое передаётся по новому сокету. Сообщение `CancelRequest` похоже на сообщение о создании сессии (**Start-up**), но только после его передачи сессия не создаётся, а соединение сервером немедленно закрывается, не возвращая результата. Аутентификация клиента не производится, для выполнения запроса достаточно правильных `pid+secret`. Любая программа, знающая `pid+secret` может прерывать запросы, это добавляет как гибкости, так и уменьшает защищённость.

Чтобы прервать выполнение запроса, клиент должен установить новое соединение к серверу и отправить ему сообщение `CancelRequest`, вместо `StartupMessage`, передаваемого при создании сессии. Сервер обработает полученное сообщение и немедленно закроет соединение, в котором передано сообщение `CancelRequest`. По соображениям безопасности, сервер закрывает соединение и не возвращает результат. Если `pid+secret` верны, то результат прерывания запроса будет виден в исходной сессии - запрос выдаст ошибку.

Сообщение `CancelRequest` может и не подействовать, если запрос успеет выполниться.

# Синхронная передача запросов

- Клиенты могут посылать запросы синхронно, используя:
  - простой протокол
    - без параметров, строкой
    - с параметрами (переменными привязки)
  - расширенный протокол с подготовкой команды
  - если есть параметры, на них ссылаются `$1`, `$2`, ...

```
psql -c "drop table t cascade; begin; set transaction read only; create table t
(c text); commit; begin; select * from t; commit;"
DROP TABLE BEGIN SET
ERROR:  cannot execute CREATE TABLE in a read-only transaction
psql -c "drop table t cascade; begin; create table t (c text); commit; begin;
select * from t; commit;"
DROP TABLE BEGIN CREATE TABLE COMMIT BEGIN
 c
---
(0 rows)
COMMIT
```



## Синхронная передача запросов

Библиотека `libpq` содержит функции, используя которые клиентские программы могут передавать серверному процессу запросы и принимать результаты. Список функций, типов данных и макросов находится в файле заголовка `libpq-fe.h`. Клиентские программы, написанные на языке C, должны включать этот файл (`#include "libpq-fe.h"`) и линковаться с библиотекой `libpq.so` (https://docs.tantorlabs.ru/tdb/ru/18_3/se/libpq-build.html )

Клиенты могут посылать запросы:

1) синхронно (https://docs.tantorlabs.ru/tdb/ru/18_3/se/libpq-exec.html ) функцией PQexec. Посылается один или несколько запросов, разделённых ";" Несколько запросов будут выполняться в одной неявной транзакции, если только не используются команды `BEGIN/COMMIT`. Если один из запросов завершается сбоем, то обработка строки запроса на этом останавливается, и возвращается описание ошибки. Пример:

```
psql -c "drop table t cascade; begin; set transaction read only; create table
t (c int); commit; begin; select * from t; commit;"
DROP TABLE BEGIN SET
ERROR:  cannot execute CREATE TABLE in a read-only transaction
```

Без ошибки все команды выполняются:

```
psql -c "drop table t cascade; begin; create table t (c int); commit; begin;
select * from t; commit;"
DROP TABLE BEGIN CREATE TABLE COMMIT BEGIN
 c
---
(0 rows)
COMMIT
```

2) синхронно с передачей переменных привязки отдельно от запроса функцией PQexecParams. Это позволяет обойтись без кавычек и экранирующих символов. В отличие от PQexec, PQexecParams может передать только один запрос. Для явного указания типов данных переменных привязки рекомендуется указывать их **тип**:

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

Обе функции используют простой протокол, то есть не подготавливают запросы к выполнению.

3) Синхронно, с подготовкой запроса функцией PQprepare. Есть много функций для работы с подготовленными запросами.

# Асинхронная и конвейерная передача запросов

- Кроме синхронной передачи можно передавать запросы асинхронно и в режиме конвейера
- Конвейерный режим (с 14 версии PostgreSQL):
  - даёт преимущества, когда сетевая задержка (ping) сравнима с длительностью выполнения запроса
  - не работает с командой COPY
  - нельзя передавать одной строкой несколько запросов
  - бесполезен когда результат одного запроса требуется для выполнения следующего запроса
  - реализовать выполнение команд в разных транзакциях сложно
  - только асинхронно и только по расширенному протоколу

```
postgres=# \bind 'abc'
postgres=# insert into t values ( $1) returning *;
  c
-----
 abc
(1 row)
INSERT 0 1
```

```
Parse("INSERT VALUES
tab(name) VALUES ($1)")
Bind("alice")
Execute()
Bind("bob")
Execute()
Sync
```



## Асинхронная и конвейерная передача запросов

4) Асинхронно с подготовкой PQsendQuery или без подготовки PQsendPrepare. Синхронное выполнение блокирует поток (thread процесса) клиентского приложения до тех пор, пока команда не начнёт выдавать результат. Отменить выполнение команды приложению затруднительно, так как его поток заблокирован.

5) В конвейерном режиме, начиная **с 14 версии** PostgreSQL. В этом режиме сразу несколько запросов/результатов могут быть отправлены/получены в одном сетевом вызове.

Конвейерный режим даёт значительный прирост производительности, если сетевая задержка велика, но писать приложения, использующие этот режим, гораздо сложнее, так как требуется реализовывать очереди ожидающих запросов и сопоставлять результаты с запросами. Для конвейерного режима, обычно, нужно больше памяти как на клиенте, так и у серверного процесса.

**В конвейерном режиме запросы передаются только асинхронно и только с использованием расширенного протокола**. Передавать несколько SQL-команд одной строкой нельзя. Также нельзя передавать команду `COPY`.

Конвейерный режим даёт преимущества, когда сетевая задержка (`ping`) велика и, когда нужно выполнить много коротких (относительно `ping`) по времени выполнения и по размеру (текст команды и полученных результатов) запросов.

Конвейерный режим бесполезен когда результат одного запроса требуется для выполнения следующего запроса.

Если команды должны выполняться в разных транзакциях, то конвейерная передача команд существенно усложняется.

Если в конвейерном режиме команда (например, **`Bind("alice")`**) вызовет ошибку (например, нарушение ограничения целостности), то происходит откат транзакции (или подтранзакции) и сообщения игнорируются до получения **`Sync`**.

При использовании в `psql` команды `\bind` используется расширенный протокол. В `psql` можно посылать команды конвейером, используя команды `\startpipline \sendpipeline \endpipeline` и другие.

https://docs.tantorlabs.ru/tdb/ru/18_3/se/libpq-pipeline-mode.html
https://postgrespro.ru/docs/postgresql/18/protocol-flow

# Аутентификация

- Для методов, где передача аутентифицирующих данных не нужна (trust, peer), сервер сразу отвечает сообщением, что соединение установлено: 2 сетевых пакета в один roundtrip
- Для MD5 один запрос и один ответ: 4 сетевых пакета в 2 roundtrips
- Для SCRAM-SHA-256 6 пакетов в 3 roundtrips
- Для остальных протоколов (kerberos, scram+ssl) число сетевых пакетов больше
- Аутентификация MD5 и SCRAM используют salt при передаче пароля по сети и защищают от перехвата пароля, но не защищают от man in the middle



## Аутентификация

Методы аутентификации GSSAPI, SSPI, SASL могут потребовать выполнить серию обменов пакетами (минимум 6 пакетов в 3 roundtrips).

Для остальных методов аутентификации может быть максимум один запрос и один ответ (итого в сумме 4 сетевых пакета в 2 roundtrips).

Для методов, где передача аутентифицирующих данных не нужна (trust, peer), сервер сразу отвечает сообщением, что соединение установлено (2 сетевых пакета в один roundtrip).

Чтобы начать сессию, клиент открывает сокет и передаёт стартовое сообщение. В этом сообщении содержатся: **имя пользователя** и **название базы данных** и версия протокола.

Сопоставив имя пользователя и базы данных с `pg_hba.conf`, `pg_ident.conf`, серверный процесс отправляет одно из сообщений:

1) `ErrorResponse` - отказ в доступе и закрывает сокет

2) `AuthenticationOk` (Аутентификация пройдена), например, при аутентификации trust

3) `AuthenticationMD5Password` (Аутентификация с паролем в формате MD5) и случайное число (соль, salt). Клиент должен передать:
`concat('md5', md5(concat(md5(concat(password, username)), random-salt))).`

Пароль в формате MD5 хранится без salt, что позволяет использовать rainbow tables (заранее вычисленные значения) для восстановления пароля, если известен хэш. Хранится хэш от "`пароль||имя`", поэтому rainbow tables создавать сложнее: только для конкретного имени пользователя, например, postgres.

4) `AuthenticationSASL`. Возможна аутентификация: SCRAM-SHA-256, SCRAM-SHA-256-PLUS. **С 18 версии** PostgreSQL и с версии 17.5 Tantor Postgres добавился OAUTH.

SCRAM-SHA-256 (RFC 7677) и его вариация с SSL(TLS) SCRAM-SHA-256-PLUS (RFC 5802) аутентифицируют по паролю.

В SCRAM без TLS серверный процесс передаёт случайное число для смешивания с паролем. Это препятствует создание новой сессии злоумышленником, перехватившим трафик. Однако, не защищает от man-in-the-middle, когда злоумышленник после прохождения аутентификации, блокирует трафик клиента и передаёт команды от себя.

SCRAM с TLS ("channel binding", сервер подтверждает себя клиенту сертификатом TLS) предотвращает такие man in the middle (MTM) атаки тем, что клиент подмешивает подпись сертификата сервера перед передачей хэша пароля. Злоумышленник не сможет установить TLS-соединение отдельно с клиентом и сервером и дешифровать трафик.

Пароль в формате SCRAM хранится с salt и rainbow tables использовать нельзя.

# Разница между MD5 и SCRAM

- по умолчанию, хэш пароля создаётся с **salt**:

```
set password_encryption = "scram-sha-256";
create user  lice password '===a';
create user alice password '===';
select rolname, left(rolpassword,60) from pg_authid where rolname like '%lice';
 lice    | SCRAM-SHA-256$4096:h+FwpVPOpkdcXbeUSTUjWg==$LN/u4HzvwJyttorD
 alice   | SCRAM-SHA-256$4096:aIFNXPHF93wZlaeYoMg2mA==$vbjjjbANZc7taTeC
```

- при использовании md5 хэш пароля хранится без **salt**:

```
drop user if exists lice, alice;
set password_encryption = md5;
create user  lice password '===a';
create user alice password '===';
select rolname, rolpassword from pg_authid where rolname like '%lice';
 lice    | md55e5043818345978ae73cd6ca5d3f76e4
 alice   | md55e5043818345978ae73cd6ca5d3f76e4
select md5('===alice');
 5e5043818345978ae73cd6ca5d3f76e4
```



## Разница между MD5 и SCRAM

При аутентификации MD5 передаётся случайное число (соль, salt). Клиент должен передать: `concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`.

Пароль в формате MD5 хранится без salt:

**`md5(concat(password, username)`**

что позволяет использовать rainbow tables для восстановления пароля, если известен хэш. Хранится хэш от "`пароль||имя`", поэтому rainbow tables создавать сложнее: только для конкретного имени пользователя, например, postgres.

Пароль в формате SCRAM хранится с salt и rainbow tables использовать нельзя.

Основное отличие MD5 от SCRAM - в сложности восстановления пароля. Часто, один и тот же пароль используется для других программ, поэтому, возможность восстановления пароля привлекательно для злоумышленников.

Формат SCRAM хранения пароля:

**`digest$iterations:salt`**`$stored_key:server_key`

По умолчанию: `digest=SCRAM-SHA-256`, `iterations=4096`, **`salt`** - случайное число размером 16 байт в формате base64.

Строки хранятся в общей таблице `pg_authid`.

**Если у клиента имеются строки MD5 или SCRAM, то он сможет аутентифицироваться, не зная пароля, только имея эти строки.**

Однако, если в строках **`salt`** разный, то аутентифицироваться не удастся. Сменив пароль на тот же самый, будет создана строка с другим **`salt`** и те клиенты, у кого строки с прежним **`salt`**, перестанут аутентифицироваться. Другими словами, если есть подозрение, что злоумышленник похитил строки SCRAM из таблицы `pg_authid` или представления `pg_shadow`, то достаточно поменять пароли на те же самые и злоумышленник не сможет аутентифицироваться. У MD5 нет **`salt`** и, чтобы похититель строки с хэшем MD5, перестал аутентифицироваться, придётся сменить пароль на другой.

# Причины использования пулов и балансировки

- Уменьшить частые запуски и остановки серверных процессов из-за коротких сессий
- Снимать с PostgreSQL нагрузку по аутентификации клиентов и установлению TLS-соединений
- Не запускать серверные процессы до аутентификации
- Не разрывать соединения при перезапуске экземпляра
- Перенаправлять запросы на текущий мастер
- Единая точка входа на несколько реплик

Виды балансировщиков:

- клиентская балансировка библиотеки libpq
- пулеры соединений для протокола PostgreSQL
- универсальные балансировщики на уровне сокетов TCP



## Причины использования пулов и балансировки

Причины использования пулов сессий и балансировки нагрузки:

1) Приложение часто создаёт сессии, выполняет несколько команд и закрывает соединение. Затраты на частое порождение серверных процессов в операционной системе впустую расходует ресурсы памяти и процессора. При завершении сессии клиентом, соединение может возвращаться в пул соединений и выдаваться другому клиенту, запрашивающему соединение. **Для каждого имени пользователя поддерживается свой пул соединений.** Задержки и трудоёмкость создания сессии зависит от способа аутентификации. Для TLS задержка наиболее существенна ~0.1 секунды. PgBouncer может подсоединять клиентов, используя TLS, а с экземплярами PostgreSQL использовать нешифрованные соединения и более простую аутентификацию, разгружая экземпляр PostgreSQL.

2) Серверные процессы запускаются до аутентификации клиента. На запуск процесса тратится больше ресурсов, чем тратит клиент на посылку сетевого пакета. Это означает, что PostgreSQL уязвим для атак типа "отказ в обслуживании" (DDOS). Балансировщик может выдавать клиенту соединение после аутентификации и не тратить ресурсы на клиентов, которые не аутентифицировались.

Клиенты могут передавать сообщения слишком медленно или слишком быстро (большие объемы данных без подтверждения приёма). Такое может произойти из-за недостатка ресурсов на клиенте, потерям сетевых пакетов. PgBouncer защищает PostgreSQL от таких клиентов и имеет параметры для точной настройки защиты: `client_login_timeout`, `max_packet_size`, `listen_backlog`, `sbuf_loopcnt`, `tcp_user_timeout` и другие.

3) При рестарте экземпляра серверные процессы останавливаются и сессии разрываются. Балансировщик не разрывает сессии и для клиента может симулировать продолжение работы в сессии, но уже с другим серверным процессом.

4) Если экземпляр останавливает обслуживание и мастером становится другой экземпляр, балансировщик может перенаправить соединения клиентов на адрес текущего мастера.

5) Если есть несколько реплик, то балансировщик может являться единой точкой входа и направлять запросы на работоспособные реплики.

Есть:

1) клиентская балансировка на уровне клиент-серверного протокола PostgreSQL

2) пулеры, которые работают на уровне протокола PostgreSQL. Пример: PgBouncer

3) балансировщики нагрузки, работающие на уровне сокетов TCP/IP. Пример: HAproxy.

# Балансировка на клиенте

- появилась в 16 версии PostgreSQL
- Параметры соединения передаются в виде:
  - строки `ключ=значение`, разделённые пробелами

```
psql --host=127.0.0.1,localhost load_balance_hosts=random --port=5432,5432
psql "host=127.0.0.1,localhost load_balance_hosts=random port=5432,5432"
```

  - URI `postgres[ql]://[пользователь@][хост][:порт][,[хост][:порт][/база_данных][?параметр=значение][&параметр=значение]`

```
psql "postgres://:5433,:5434/postgres?target_session_attrs=primary&load_balance_hosts=random"
```

- в терминале, если в строке есть символ "&", строку нужно заключить в " " или ` ` или ' '
- Подсоединение через UNIX-сокет:

```
postgres://%2Fvar%2Frun%2Fpostgresql:5433/postgres?options=-c%20geqo%3Doff
```



## Балансировка на клиенте

Клиентская балансировка в сетевой библиотеке libpq появилась **в 16 версии**:

```
psql --host=tantor,localhost load_balance_hosts=random --port=5432,5432
psql "host=tantor,localhost load_balance_hosts=random port=5432,5432"
```

Параметры соединения передаются в библиотеку `libpq` в виде:

1) простой строки `ключ=значение`, разделённых пробелами. Значения могут заключаться в апострофы. Символы апострофа и "\" предваряются знаком "\".

2) URI в виде `postgres[ql]://[пользователь@][хост][:порт][,[хост][:порт][/база_данных][?параметр=значение][&параметр=значение]`

Знак "=" можно заменить на "`%3D`", знак пробела на "`%20`". Примеры:

Подсоединение через UNIX-сокет:

```
postgres://%2Fvar%2Frun%2Fpostgresql:5433/postgres?options=-c%20geqo%3Doff
```

URI вида `postgresql://host1:port1,host2:port2,host3:port3/` равнозначно строке подключения вида `host=host1,host2,host3 port=port1,port2,port3`. Хосты будут перебираться по порядку, пока не будет установлено соединение. Если используется файл паролей, в нём можно задать разные пароли для разных узлов. Все остальные параметры подключения будут одинаковыми для всех узлов; например, нельзя задать для разных узлов различные имена пользователей.

```
postgres://:5433,:5434/postgres?target_session_attrs=primary
```

При передаче строки соединения утилитам командной строки в терминале, если в строке есть символ "&", то строку придётся заключить в двойные кавычки или обратные кавычки или апострофы: `psql "postgres://:5433,:5434/postgres?target_session_attrs=primary&load_balance_hosts=random"`

Балансировка на уровне клиента есть не только в библиотеке libpq, но и в других драйверах: jdbc, npgsql (C#), rust-postgres.

Пулы соединений на уровне драйверов есть в спецификации JDBC: javax.sql.ConnectionPoolDataSource.

https://docs.tantorlabs.ru/tdb/ru/18_3/se/libpq-connect.html#LIBPQ-CONNECT-LOAD-BALANCE-HOSTS

# Параметры для балансировки на клиенте

- `load_balance_hosts` - порядок перебора адресов:
  - `disable` (по умолчанию) - в указанном порядке
  - `random` - в случайном порядке
- `target_session_attrs` - соединение устанавливается с первым, подходящим под условия, хостом. Значения:
  - `any` (по умолчанию) - без ограничений
  - `primary` - с мастером (`in_hot_standby=off`)
  - `standby` - с физической репликой (`in_hot_standby=on`)
  - `read-write` - принимает пишущие транзакции (in_hot_standby=`off` и default_transaction_read_only=`off`)
  - `read-only` - обратное `read-write`
  - `prefer-standby` - сначала попытаться найти реплику



## Параметры для балансировки на клиенте

Для клиентской балансировки используются два параметра. Параметрами можно указать, что соединения должны направляться на мастер или одну из реплик. Обычно, с параметрами используется несколько значений host и port, но можно использовать и/или DNS-балансировку.

Первый параметр:

`load_balance_hosts` - порядок, в котором клиент пытается подключиться к доступным адресам. Если попытка подключения окажется успешной, клиент не будет пытаться подключиться к другим адресам. Параметр можно использовать вместе с `target_session_attrs`, например для балансировки нагрузки только между резервными серверами. Два режима:

`disable` (по умолчанию) - адреса перебираются в указанном порядке

`random` - адреса перебираются в случайном порядке и сессии распределяются по адресам случайно. Если нужно, чтобы какой-то адрес принимал больше соединений, то его можно указать несколько раз. И обратно: если в DNS или в файле `/etc/hosts` указать несколько IP-адресов, то сессии будут случайно распределяться между ними.

Второй параметр:

`target_session_attrs` - условия соединения. Соединение устанавливается с первым, подходящим под условия, хостом. Значения:

`any` (по умолчанию) - без ограничений

`primary` - кластер является мастером (`in_hot_standby=off`)

`standby` - кластер является физической репликой (`in_hot_standby=on`)

`read-write` - принимает пишущие транзакции (`in_hot_standby=off` и `default_transaction_read_only=off`)

`read-only` - обратное `read-write`

`prefer-standby` - сначала пытаться найти реплику, но если ни один из перечисленных кластеров не является репликой, повторить в режиме `any`.

# Ограничения клиентской балансировки libpq

- Используется перебор соединений, что создаёт задержки на установку соединения
  - Стоит установить параметр `connect_timeout`
- Не анализируется отставание реплики от мастера
- Смена роли обнаруживается с задержкой, если закрывается сетевое соединение или в ответ на команду получена ошибка



## Параметры для балансировки на клиенте

Ограничения клиентской балансировки libpq:

1) Клиентская балансировка libpq определяет роль экземпляра подсоединяясь к нему и выполняя функцию `pg_is_in_recovery()`. Заранее определить роль нельзя, поэтому libpq перебирает соединения. Перебор соединений создаёт задержки, так как при аутентификации, а ещё хуже, при TLS-соединениях передаётся много сообщений.

Рекомендуется установить параметр `connect_timeout`. По умолчанию, значение параметра не установлено, что означает, что ожидание ответа сервера бесконечно. Таймаут действует на каждый адрес. Если адресов 3, таймаут 10, то максимальное ожидание установки соединения может достигать 30 секунд.

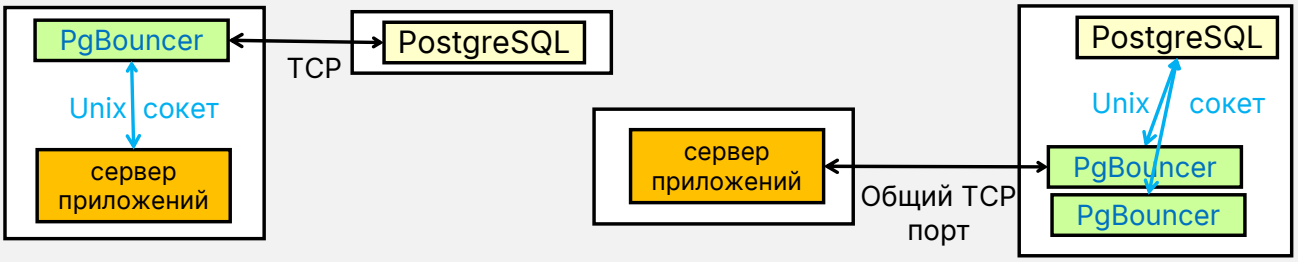
Для устранения задержек на установку соединения можно использовать пулеры соединений.

2) Не анализируется отставание реплики от мастера. Клиент будет подключен к первой ответившей реплике.

3) Уведомлений о смене роли реплика-мастер клиенту не посылается.

# PgBouncer

- Однопроцессный, однопоточный пулер соединений и балансировщик нагрузки
- Вносит минимальные задержки
- Запускается как служба, прослушивает один порт
- Прозрачен для клиентов и экземпляров PostgreSQL
- Работает по протоколам TCP и Unix-сокет
- Уменьшает издержки на создание сессий
- Расположение процесса `pgbouncer`:






## PgBouncer

- однопроцессный, однопоточный **пулер соединений** с элементами балансировки нагрузки;
- написан на языке C, один исполняемый файл, использует библиотеку **`libevent`**, которая вносит минимальные задержки и позволяет создавать высокопроизводительные сетевые приложения;
- работает по тем же протоколам, что и PostgreSQL: TCP и Unix-сокеты;
- может запускаться на хосте с экземпляром PostgreSQL или любом другом хосте;
- прозрачен для клиентов: выглядит как сервер PostgreSQL;
- для экземпляра PostgreSQL выглядит как набор обычных клиентов;
- располагается между клиентами и серверными процессами, удерживая с ними сокеты, периодически подключая и отключая пары клиент-сервер в соответствии с используемым видом пула. На каждое соединение выделяет пару килобайт, может обслуживать тысячи соединений, вносит минимальные сетевые задержки.
- запускается как служба, прослушивает один порт. Реализует пулы соединений с экземплярами PostgreSQL. Клиент обращается на порт PgBouncer, который аутентифицирует клиента и направляет команды клиента на существующее соединение с базой данных PostgreSQL или создаёт новое соединение.

Основная задача - уменьшить издержки на порождение и остановку серверных процессов, позволяя клиентам повторно использовать соединения с серверными процессами.

# PgBouncer: пул сессий

- Три режима работы пулов
- Первый режим:
  - session - клиенту назначается соединение с одним и тем же серверным процессом, пока клиент не закроет сессию
- Преимущества session пулов:
  - серверные процессы не перезапускаются при частых переподсоединениях клиентов;
  - обрабатывает аутентификацию, снимая нагрузку с серверных процессов
  - при перезапуске экземпляров соединения не разрываются
  - для клиента нет различий с прямым соединением



## PgBouncer: пул сессий

Вид пула устанавливается параметром `pool_mode`, который можно установить глобально или на уровне описателя базы данных.

Имеет три вида пулов:

Пулы типа session. Клиенту назначается соединение с одним и тем же серверным процессом, пока клиент не закроет сессию. При закрытии сессии клиентом, соединение возвращается в пул и выдаётся любому клиенту, запрашивающему новое соединение.

Преимущества этого вида пула:

1) серверные процессы не перезапускаются при частых переподсоединениях клиентов;

2) обрабатывает аутентификацию, если она трудоемка (много сообщений как в SCRAM, при больших сетевых задержках);

3) может удерживать сессии с клиентами при перезапуске экземпляров;

4) для клиента нет различий с прямым подсоединением к серверному процессу.

# PgBouncer: пул транзакций

Другие два режима работы пулов:

- transaction - соединение назначается клиенту на время транзакции
  - если большую часть времени сессии простаивают, то число серверных процессов может быть существенно меньше числа сессий
  - отслеживание подготовленных запросов включено по умолчанию
  - не сохраняются: `SET/RESET`, `PREPARE/DEALLOCATE`, сессионные рекомендательные блокировки, временные таблицы, `LISTEN`, курсоры `WITH HOLD`
  - библиотеки, загруженные командой `LOAD`
- statement - нельзя использовать транзакции, только одиночные команды



## PgBouncer: пул транзакций

Пулы типа transacion. Соединение назначается клиенту только на время транзакции. Разные транзакции могут обслуживаться разными серверными процессами. Между транзакциями не сохраняются:

1) параметры конфигурации на уровне сессии (`SET/RESET`)
2) подготовленные команды (`PREPARE/DEALLOCATE`)
3) рекомендательные блокировки на уровне сессии
4) временные таблицы (кроме `ON COMMIT DROP`)
5) `LISTEN` (`NOTIFY` возможен)
6) курсоры `WITH HOLD`
7) библиотеки, загруженные командой `LOAD` в память серверного процесса.

PgBouncer может выполнить команды (например, команды подготовки запросов) перед установкой соединений с экземпляром.

В этом режиме добавляется преимущество: меньшее число серверных процессов могут обслуживать большое число сессий клиентов. Если большую часть времени сессии простаивают, то число серверных процессов может быть существенно меньше числа сессий.

Начиная с версии 1.21 (октябрь 2023г.) PgBouncer, на уровнях **transacion** и **statement** подготовленные запросы отслеживаются и автоматически подготавливаются, за исключением явных команд `PREPARE/EXECUTE/DEALLOCATE`. Запросы, подготовленные SQL-командой `PREPARE` не отслеживаются.

Начиная с версии 1.24 (01.2025г.) отслеживание подготовленных запросов включено по умолчанию: параметр PgBouncer `max_prepared_statements` установлен в значение `200`.

Если приложение протестировано разработчиками для работы с пулами **transaction** и работает корректно, то стоит использовать этот режим вместо режима **session**. Режим **transaction** позволяет приложению создавать большое число сессий с PgBouncer, которые будут обслуживаться небольшим числом сессий с экземплярами PostgreSQL.

Пулы типа **statement**. Соединение возвращается в пул после выполнения одного запроса. Транзакции в этом режиме нельзя использовать, только режим автофиксации. Этот режим не имеет преимуществ (бесполезен).

Ограничения:
https://www.pgbouncer.org/features.html
Новшества:
https://www.pgbouncer.org/changelog.html

# Подготовленные запросы в пулах транзакций

- `max_prepared_statements` по умолчанию 200
  - желательно устанавливать так, чтобы он был больше, чем число подготовленных запросов, используемых приложением
- PgBouncer отслеживает сообщение `Prepare()` клиент-серверного протокола и подготавливает поименованные запросы в тех сессиях, где используются эти запросы
- Команды SQL `PREPARE foo AS` и `EXECUTE foo` не отслеживаются, они выполняются в сессии, с которой работает клиент в момент выполнения команды
  - команды могут встречаться в скриптах psql



## Подготовленные запросы в пулах транзакций

`max_prepared_statements` по умолчанию 200, начиная с версии 1.24 BgBouncer. Если значение отлично от нуля, то PgBouncer отслеживает сообщения расширенного протокола.

PgBouncer перехватывает сообщение `Prepare()` клиент-серверного протокола от клиента, присваивает запросу внутреннее имя `PGBOUNCER_`*`NNNN`* вместо имени запроса, заданного клиентом, и сохраняет сопоставление этих имён **и текст самого запроса** в своей памяти. Когда тот же или любой другой клиент, позже посылает сообщения `Bind/Execute` на этот запрос через любое соединение с серверным процессом экземпляра PostgreSQL, то PgBouncer проверяет, есть ли в сессии с серверным процессом этот запрос под внутренним именем. Если есть, то запрос выполняется. Если нет, PgBouncer посылает серверному процессу команду сообщение подготовить запрос и перенаправляет сообщения `Bind/Execute`. Параметр `max_prepared_statements` определяет число подготовленных запросов, которые остаются активными в кеше LRU PgBouncer в каждом соединении с серверным процессом. В PostgreSQL параметра с таким названием нет. Значение `max_prepared_statements` желательно устанавливать так, чтобы он был больше, чем число подготовленных запросов, используемых приложением.

Более высокое значение параметра требует больше памяти на сопоставление запросов в памяти PgBouncer.

С отслеживанием может возникать проблема. Если разные клиенты подготовили одинаковые тексты запросов, но используют разные типы данных для параметров подготовленного запроса или столбцов результатирующей выборки, то будет выдаваться ошибка:

```
ERROR:  cached plan must not change result type
```

Убрать ошибку, выполнив повторную подготовку запроса, можно командой RECONNECT в базе pgbouncer.

Команды SQL `PREPARE foo AS` и `EXECUTE foo` не отслеживаются, они выполняются только в одной сессии. PgBouncer анализирует сообщения клиент-серверного протокола, но анализирует команды SQL. Обычно, приложения используют методы/функции клиентского драйвера и не посылают SQL команды PREPARE, но в скриптах, выполняемых в psql, эти команды могут встречаться.

Драйвера psycopg3, JDBC, pgx (язык Go) работают корректно, PHP с версии 8.4 и libpq 17.

# Состояние сессий в пулах

- В пулах **сессий**, при возврате сессии в пул, выполняется команда из параметра `server_reset_query`, по умолчанию, `DISCARD ALL`, эквивалентная командам:

```
CLOSE ALL;
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
DISCARD SEQUENCES;
```

- В пулах транзакций выполняется, если установить `server_reset_query_always=1`
- Выгрузка и загрузка дампов работает некорректно на пулах транзакций
- работоспособность сессий может проверяться:
- `server_check_query` - текст проверочного запроса
- `server_check_delay` - интервал проверки



## Состояние сессий в пулах

В пулах **сессий**, при возврате сессии в пул, выполняется команда из параметра `server_reset_query`, по умолчанию, `DISCARD ALL`. Эта команда очищает всё, что может сохраняться в сессии. В пулах транзакций `server_reset_query` выполняется, если установить `server_reset_query_always=1` (по умолчанию, 0 - отключено). Накладных расходов команда `DISCARD ALL` не даёт.

Если в пуле транзакций клиент меняет значения параметров сессии, например, `SET statement_timeout = '5s'`, то следующий клиент, который получит эту сессию унаследует 5-секундный таймаут и его запросы будут прерываться неожиданно для клиента, что будет выглядеть как сбой.

Дампы, восстанавливаемые утилитами `psql` или `pg_restore`, устанавливают командами `SET` набор параметров и ожидается, что параметры продолжат действовать. Выгрузка и загрузка дампов работает некорректно на пулах **транзакций**.

PgBouncer перед выдачей сессий из пулов клиентам, может проверять работоспособность сессий параметрами:

1) `server_check_query` - текст тестового запроса. По умолчанию:

`server_check_query = <empty>`

для проверки посылается пустая команда и планировщик не задействуется.

Если вместо `<empty>` установить пустую строку:

`server_check_query =`

то проверка не будет выполняться.

Писать запросы нет смысла, так как pgbouncer не анализирует выдал ли запрос ошибку на SQLуровне или не выдал, анализируется только пришёл ли любой сетевой пакет от сервера или не пришёл. Если установить `select 1`, то будет выполняться планирование запроса.

2) `server_check_delay` - по умолчанию, 30 секунд. Задаёт свежесть соединения, после тестового запроса, в течение которого сессия может передаваться клиенту из пула без проверки. Ноль не стоит ставить, так как при значении 0, тестовый запрос запускается перед передачей сессии клиенту, что вносит существенную задержку.

Если клиент не может получить сессию из пула дольше `query_wait_timeout` (по умолчанию, 2 минуты), то его сессия будет отсоединена.

# PgBouncer: общий порт

- Можно запустить несколько процессов pgbouncer на одном порту TCP (параметр `so_reuseport=1`)
- Используется для "rolling update" - бесшовного перезапуска при смене значений параметров или версии PgBouncer
- Если используется общий порт, у процессов pgbouncer должны быть разные:
  - директории Unix-сокета (`unix_socket_dir`)
  - файлы `pidfile` и `logfile` и файлы параметров



## PgBouncer: общий порт

Число соединений фактически ограничивается (~тысячи соединений) производительностью одного ядра процессора, так как PgBouncer однопоточен.

Однако, можно запустить несколько процессов PgBouncer, даже на одном порту, используя параметр `so_reuseport=1`. При установке параметра, серверный TCP-сокет открывается с опцией `SO_REUSEPORT`. Общий порт позволяет запускать несколько процессов PgBouncer на одном хосте, чтобы они прослушивали один и тот же TCP-порт. Запросы клиентов на порт будут автоматически распределяться по процессам ядром linux. Это позволяет обойти ограничение однопоточности процесса PgBouncer и задействовать несколько ядер процессора.

Используя один порт, число эффективно обрабатываемых соединений (без существенной задержки) - десятки тысяч (до ~40000). Процессы PgBouncer могут запускаться на одном порте или на разных портах того же или разных хостов.

**Если используется общий порт, то директории для файла Unix-сокета (`unix_socket_dir`) у процессов PgBouncer должны быть разными**. Один файл Unix-сокета не может использоваться несколькими процессами.

**Процессы PgBouncer должны использовать разные файлы `pidfile` и `logfile` и файлы параметров**.

При использовании общего порта для подключения к Admin console (псевдобаза pgbouncer) конкретного процесса PgBouncer нельзя подключиться по TCP/IP, через Unix-сокет можно.

При использовании общего порта нужно использовать пиринг, иначе прерывание выполнения запросов не будет корректно работать.

# PgBouncer: пиринг

- Если несколько процессов PgBouncer обслуживает тех же клиентов, нужно сконфигурировать секцию `[peers]`
- у каждого PgBouncer добавляется его идентификатор в виде числа: у первого `peer_id=1`, у второго `peer_id=2`
- Без пиринга, не смогут выполняться **Cancel request**
- Пример `/opt/tantor/etc/pgbouncer/pgbouncer2.ini`:

```
[pgbouncer]
pidfile = /var/run/pgbouncer-tantor/pgbouncer2.pid
logfile = /opt/tantor/var/log/pgbouncer/pgbouncer2.log
listen_port = 6432
unix_socket_dir = /var/run/pgbouncer-tantor/2
so_reuseport=1
peer_id = 2
[peers]
1 = host=/var/run/pgbouncer-tantor/1 port=6432
2 = host=/var/run/pgbouncer-tantor/2 port=6432
```



## PgBouncer: пиринг

Если запускается несколько процессов PgBouncer, к которым подсоединяются одни и те же клиенты, то в параметры конфигурации процессов нужно добавить параметры в секцию `[peers]`. В секцию вставляются адреса PgBouncer, входящих в группу.

Такая конфигурация используется, если между клиентами и процессами PgBouncer используется балансировщик нагрузки или DNS-балансировка или балансировка libpq, а также, если используется общий порт.

Пример конфигурации второго процесса PgBouncer для двух процессов:

```
[pgbouncer]
pidfile = /var/run/pgbouncer-tantor/pgbouncer2.pid
logfile = /opt/tantor/var/log/pgbouncer/pgbouncer2.log
listen_port = 6432
unix_socket_dir = /var/run/pgbouncer-tantor/2
so_reuseport=1
peer_id = 2
[peers]
1 = host=/var/run/pgbouncer-tantor/1 port=6432
2 = host=/var/run/pgbouncer-tantor/2 port=6432
```

у каждого PgBouncer в его конфигурационный файл добавляется его идентификатор в виде числа: у первого `peer_id=1`, у второго `peer_id=2`. По умолчанию, `peer_id=0` (пиринг отключён).

Без конфигурирования пиринга, **Cancel request** не смогут выполняться. Проблема в том, что клиенты передают сообщение **Cancel request** в отдельном соединении, похожем на Start-up. Это соединение может попасть на произвольный процесс PgBouncer. С пирингом, PgBouncer получивший сообщение, сможет передать сообщение (тоже по отдельному сокету) тому процессу PgBouncer. в сессии которого выполняется запрос, который клиент хочет прервать.

Общий порт TCP/IP может использоваться совместно с пирингом и можно установить значение `so_reuseport=1`. Однако, сообщения **Cancel request** должны передаваться на адрес конкретного PgBouncer. У процессов PgBouncer должны быть разные директории Unix-сокетов. В секции `[peers]` нужно указать эти директории Unix-сокетов.

Максимальное время ожидание выполнения **Cancel request** задано параметром `cancel_wait_timeout` (по умолчанию, 10 секунд).

# Установка PgBouncer

- Можно установить скриптом:

```
wget https://public.tantorlabs.ru/db_extension_installer.sh
chmod +x db_extension_installer.sh
export NEXUS_URL="nexus-public.tantorlabs.ru"
./db_extension_installer.sh --database-type=tantor --edition=all
--extension=pgbouncer
```

- или пакетным менеджером:

```
apt install libc-ares2
dpkg -i pgbouncer-tantor-all_*.de
```

- Файлы:
- /opt/tantor/etc/pgbouncer/pgbouncer.ini конфигурационный
- /opt/tantor/etc/pgbouncer/userlist.ini
- /usr/lib/systemd/system/pgbouncer-tantor.service



## Установка PgBouncer

PgBouncer устанавливается из отдельного пакета `deb` или `rpm`.

Можно установить из репозитория Тантор скриптом установки расширений:

```
wget https://public.tantorlabs.ru/db_extension_installer.sh
chmod +x db_extension_installer.sh
export NEXUS_URL="nexus-public.tantorlabs.ru"
./db_extension_installer.sh --database-type=tantor --edition=all --
extension=pgbouncer
```

Также можно установить pgbouncer из пакета, но придётся установить пакеты, от которых зависит pgbouncer:

```
apt install libc-ares2
dpkg -i pgbouncer-tantor-all_*.deb
```

**c-ares** используется для разрешения DNS-имён вместо evdns из libevent, так как поддерживает EDNS (RFC 6891). EDNS нужен, чтобы поддерживать случаи, когда по одному имени хоста DNS выдаёт больше 8 IP адресов.

После установки, служба устанавливается, но не запускается:

```
pgbouncer-tantor.service is a disabled or a static unit, not starting it.
```

Балансировщик устанавливается в директорию `/opt/tantor`. Основные файлы:

`/opt/tantor/usr/bin/pgbouncer` - исполняемый файл

`/opt/tantor/etc/pgbouncer/pgbouncer.ini` - конфигурационный файл

**`/usr/lib/systemd/system/pgbouncer-tantor.service`** - файл с описателем службы

В файле службы указано под каким **пользователем** linux запускать процесс:

```
[Service]
Type=notify
User=pgbouncer
Group=pgbouncer
```

Пользователя и группу pgbouncer можно поменять на пользователя postgres, чтобы упростить доступ к директории с файлами Unix-сокетов и аутентификацию процесса PgBouncer в экземплярах PostgreSQL, если они работают на том же хосте.

Путь к файлу со списком пользователей и их паролями или хэшами паролей в формате `MD5` или `SCRAM` задан в параметре: `/opt/tantor/etc/pgbouncer/userlist.ini`

Путь к логу задан в параметре, файлы ротации в этой же директории:

```
logfile = /opt/tantor/var/log/pgbouncer/pgbouncer.log
```

# Параметры PgBouncer

- После установки отредактировать файл

```
[databases]
* = port=5432
[pgbouncer]
listen_addr = *
listen_port = 6432
unix_socket_dir = /var/run/pgbouncer-tantor
auth_type = md5
auth_file = /opt/tantor/etc/pgbouncer/userlist.txt
auth_dbname = template1
auth_user = postgres
admin_users = postgres, pgbouncer
server_reset_query_always = 1
server_fast_close = 1
so_reuseport = 1
```

- поменять пароли (хэши паролей) в `userlist.txt`

```
"postgres" "postgres"
"pgbouncer" "pgbouncer"
"bob" "md5e8557d12f6551b2ddd26bbdd0395465c"
```



## Параметры PgBouncer

Настройка PgBouncer - это редактирование его файла параметров.

В файле `userlist.ini` поменять пароли, которые там присутствуют на другие.

Пользователи PostgreSQL, перечисленные в этом файле, смогут подсоединяться к базам данных через PgBouncer.

Пароли можно указать текстом или в формате хэшей PostgreSQL MD5 или SCRAM. Хэши можно найти в каждом кластере PostgreSQL запросом:

```
select rolname, rolpassword from pg_authid;
```

Пользователи, не указанные в файле, не смогут подключиться через PgBouncer, если только не установить значение параметру `auth_user` (и, возможно, `auth_dbname`, `auth_query`).

Чтобы можно было давать команды PgBouncer в секции `[pgbouncer]` файла `pgbouncer.ini` указать имена пользователей, которые смогут подсоединяться к псевдобазе, имя которой pgbouncer.

```
admin_users = postgres, pgbouncer
```

Параметром `stats_users` можно задать пользователей, которые могут выполнять команды `SHOW`, при подсоединении к псевдобазе `pgbouncer`. Эта база называется "Admin console" (консоль управления) PgBouncer. Подсоединившись к этой базе можно выполнять команды управления и мониторинга PgBouncer. Команды описаны в документации. Базы с именем `pgbouncer` не должно быть в кластерах PostgreSQL, при подсоединении к базе с таким именем, можно будет давать команды управления PgBouncer и команды будет выполнять PgBouncer.

PgBouncer принимает соединения по адресу, задаваемому параметрами:

```
listen_addr = localhost
listen_port = 6432
unix_socket_dir = /var/run/pgbouncer-tantor
```

Адреса в параметре `listen_addr` можно перечислить через запятую.

В параметре `listen_addr` можно использовать звёздочку (*), что означает все IP-адреса.

# Секция [databases]

- отредактировать файл параметров:

```
[databases]
db01 = host=/var/run/postgresql port=5433 dbname=postgres pool_mode=session
db01ro = host=10.0.0.1,10.0.0.2 port=5434 dbname=postgres load_balance_hosts=round-robbin
db02 = host=127.0.0.1,localhost,::1 port=5433 dbname=abcd
* = port=5432
```

  - параметры задаются в формате:
    - описатель = параметр=значение
    - других форматов, как у libpq (postgresql:// и "строка") нет
  - Параметры не являются параметрами **libpq** и имеют смысл, описанный в документации к PgBouncer
    - параметра **libpq** target_session_attrs нет, мастер и реплики не распознаются



## Секция [databases]

В секции [databases] файла pgbouncer.ini указываются описатели "имён баз", которые смогут указывать клиенты и адреса реальных баз, куда будут направляться соединения.

Описатель со звёздочкой * в качестве "имени базы" используется, если имя базы, к которой хочет подсоединиться клиент отсутствует в секции [databases].

"Имя базы" можно заключать в двойные кавычки, если в них присутствуют символы, отличные от букв английского алфавита, цифр и знака подчёркивания.

Клиент указывает описатель "имени базы" как имя базы данных, к которой хочет подключиться, подсоединение будет установлено к базе, имя которой указывается в параметре **dbname**:

```
psql -qp 6432 -h localhost -U alice -d db01
Password for user alice:
db01=# select current_database();
 current_database
------------------
 postgres
```

Если параметр **dbname** не указан, будет использоваться имя, которое указывает клиент (**-d db01**).

В параметре host через запятую перечисляются IP адреса (V4 или V6) или имена хостов. Если имя начинается на "/" или "@", то указывается директория Unix-сокета.

Если параметр **host** не указан, то используется Unix-сокет.

**Если PgBouncer работает на том же хосте, что и экземпляр PostgreSQL, стоит использовать Unix-сокет, а не IP адрес (localhost).**

Хосты должны быть доступны. Возможности игнорировать недоступные хосты нет. По умолчанию, load_balance_hosts=round-robbin и хосты будут использоваться по кругу. Рекомендуется уменьшить значение параметра server_login_retry (по умолчанию, 15 секунд).

Если указать параметр user, то все подсоединения по "имени базы" будут подсоединяться под этим пользователем и использовать общий пул сессий, что удобно. Можно ещё указать параметр password, иначе будет использоваться пароль этого пользователя в файле auth_file.

В строке "имени базы" можно перекрыть много глобальных параметров.

# Параметр `unix_socket_dir`

- PgBouncer можно расположить как на хосте с экземпляром PostgreSQL, так и на хостах серверов приложений
- стоит поменять значение параметра `unix_socket_dir` на `/var/run/postgresql`
  - дать разрешения пользователю `pgbouncer` на эту директорию
  - или в файле службы заменить пользователя и группу **pgbouncer** на **postgres**
- или `/var/run/pgbouncer-tantor/1`
- или, если используется несколько процессов pgbouncer (пиринг), создать свою директорию для каждого процесса pgbouncer: `/var/run/pgbouncer-tantor/1` и указать её в парметре `unix_socket_dir`



## Параметр `unix_socket_dir`

Задержка на установку сессии зависит от способа аутентификации. Самая маленькая задержка у метода trust. Если сессии часто создаются, то влияние задержки существенно. PgBouncer можно запускать на хосте сервера приложений, который не имеет встроенного пулера и с которого часто порождаются сессии. Так как клиент и PgBouncer на одном хосте, то можно использовать для аутентификации метод trust и даже Unix-сокет вместо localhost, чтобы ещё больше снизить задержку. В таком случае, удобно использовать стандартную для утилит и драйверов PostgreSQL директорию `unix_socket_dir = /var/run/postgresql`. PgBouncer может подсоединяться к хосту с экземплярами PostgreSQL используя аутентификацию. Задержки на аутентификацию не будут существенны, так как PgBouncer удерживает сессии с экземплярами от 10 минут (`server_idle_timeout`) до часа (`server_lifetime`).

Если сервера приложений имеют встроенный пулер и удерживают сессии долго, то PgBouncer можно располагать на хосте с экземпляром PostgreSQL. В этом случае, PgBouncer может подсоединяться к экземпляру по Unix-сокет. Клиенты будут аутентифицироваться PgBouncer любыми способами, безопасность которых достаточна для сети между серверами приложений и хостом, где работает PgBouncer. В такой конфигурации можно использовать `unix_socket_dir = /var/run/pgbouncer-tantor/1` и общий порт для перезапуска процессов pgbouncer без разрыва сессий.

Если BgBouncer запускается под пользователем операционной системы **pgbouncer**, то он не имеет доступа на запись к директории сокетов **postgres** и директория сокета задана как `/tmp`. Если утилитам postgres нужно подсоединяться через PgBouncer по Unux-сокету, то можно поменять значение параметра в секции `[pgbouncer]` файла `pgbouncer.ini`:

```
;; On Debian it should be /var/run/postgresql
unix_socket_dir = /var/run/postgresql
```

и дать право пользователю **pgbouncer** создавать файлы в директории сокетов:

```
chown postgres:pgbouncer /var/run/postgresql
chmod 775 /var/run/postgresql
```

либо в файле службы заменить пользователя и группу **pgbouncer** на **postgres**.

Если у `unix_socket_dir` установлено пустое значение, то PgBouncer не принимает соединения по Unux-сокету.

# Параметр `server_fast_close`

- `server_fast_close=1` закрывает сессии с серверным процессом и с клиентом после завершения транзакции, если процессу pgbouncer был послан сигнал `RELOAD` (`SIGHUP`) или `RECONNECT` (`SIGTERM`)
- клиент должен будет переподсоединиться, так как его сессия будет завершена
  - приложения, обычно, рассчитаны на внезапный разрыв простаивающей сессии
- действует для пулов **session**
- на пулы **transaction** и **statement** не влияет



## Параметр server_fast_close

Для пулов **session** полезен параметр `server_fast_close=1`. Параметр позволяет закрывать сессии с серверным процессом и клиентом после завершения транзакции, если процессу pgbouncer был послан сигнал `RELOAD` (`SIGHUP`) или `RECONNECT` (`SIGTERM`). Пулы **transaction** и так возвращают соединения в пул сразу по завершению транзакции и не задерживают надолго обработку `RELOAD` и `RECONNECT`, поэтому параметр на такие пулы не влияет.

Установка параметра не приводит к потере результатов транзакций. Единственно, клиент должен будет переподсоединиться, так как его сессия будет завершена. Использование параметра `server_fast_close=1` безопасно, так как приложения, обычно, рассчитаны на внезапный разрыв простаивающей сессии. Приложения не рассчитаны на разрыв сессии в момент от передачи сообщения о фиксации транзакции и до момента получения приложением подтверждения о фиксации транзакции.

# Параметры соединений с экземплярами

- По умолчанию `server_round_robin=0` и соединения из пула отдаются клиентам в порядке LIFO, что оптимально для пула, который соединяется с одним экземпляром
- `server_lifetime` длительность сессии, по истечении которого она будет закрыта, как только вернётся в пул (по умолчанию, 1 час)
  - значение 0 заставляет закрывать сессии после использования клиентами, что нежелательно
- `server_idle_timeout` (по умолчанию, 10 минут) позволяет закрывать простаивающие сессии даже, если они назначены клиентам, но при этом, простаивают



## Параметры соединений с экземплярами

По умолчанию, PgBouncer повторно использует соединения из пула в порядке LIFO (последнее пришло, первое ушло) и активно используются несколько последних соединений из пула. Это даёт наибольшую производительность, если pgBouncer соединяется с единственным экземпляром, так как преимущественно используемые серверные процессы не вытеснены планировщиком linux как простаивающие. Если PgBouncer подсоединяется не напрямую к экземпляру, а к другому балансировщику, который мог создать сессии с разными экземплярами PostgreSQL (несколько реплик), то будут преимущественно использоваться только несколько последних сессий и возникнет перекос нагрузки, если последние сессии были созданы с одним экземпляром (мастера). Чтобы этого не было, можно установить параметр `server_round_robin=1`, тогда PgBouncer будет использовать все сессии по кругу.

Если нужно, чтобы не было долгих сессий от PgBouncer до экземпляров PostgreSQL, можно установить параметр `server_lifetime`, который задаёт длительность сессии, по истечении которого сессия будет закрыта, как только она вернётся в пул. Закрываются только неиспользуемые клиентами сессии (по умолчанию, 3600 = 1 час). Периодическое пересоздание сессий позволяет установить новое соединение с, возможно, другим экземпляром, то есть периодически перебалансировать распределение сессий по экземплярам. Значение 0 заставляет закрывать сессии после использования клиентами, что нежелательно.

Параметр `server_idle_timeout` (по умолчанию, 600 секунд) позволяет закрывать сессии даже, если они простаивают, будучи назначенными клиентам.

Аналогичные параметры для клиентских сессий, которые простаивают дольше, чем `client_idle_timeout` или `transaction_timeout` или `idle_transaction_timeout` (по умолчанию отключены - 0).

Параметр `default_pool_size` ограничивает число соединений PgBouncer для **каждой** пары пользователь+база PostgreSQL, то есть размер "пула соединений" (используемых, в каждый момент времени, сессий из пула). По умолчанию 20, что является разумным значением даже для промышленной эксплуатации.

Параметр `max_db_connections` - ограничение числа соединений от PgBouncer к каждой базе PostgreSQL, `max_db_client_connections` - ограничение числа соединений клиентов к PgBouncer по описателю одной базы. По умолчанию, оба параметра `0` (без ограничений) и менять это не стоит. Параметры можно задать на уровне описателя базы данных.

# Число соединений к PgBouncer

- `max_client_conn` - максимальное число соединений к PgBouncer, по умолчанию 100
- ограничивается числом файловых дескрипторов в linux
- Проверка ограничения у работающих процессов пользователя **pgbouncer**:

```
for PID in $(pgrep "pgbouncer"); do cat /proc/$PID/limits | grep files;
done | uniq
Max open files 1024  524288  files
```

- Для увеличения лимитов у службы добавить строки:

```
cat /usr/lib/systemd/system/pgbouncer-tantor.service
...
[Service]
LimitNOFILE=infinity
LimitNOFILESoft=infinity
...
```



## Число соединений к PgBouncer

Максимальное число соединений к PgBouncer ограничивает параметр `max_client_conn`. Значение по умолчанию 100. Ограничивается числом файловых дескрипторов linux.

Проверка ограничения у работающих процессов пользователя **pgbouncer**:

```
for PID in $(pgrep "pgbouncer"); do cat /proc/$PID/limits | grep files; done
| uniq
Max open files 1024  524288  files
```

В примере `soft limit 1024`, `hard limit 524288`. При превышении `soft limit` процесс получает предупреждение. При превышении `hard limit` процесс не сможет открыть новые файлы, пока не будут закрыты ранее открытые файлы. Для увеличения лимитов у службы нужно добавить в секцию `[Service]` строки:

```
cat /usr/lib/systemd/system/pgbouncer-tantor.service
...
[Service]
Type=notify
User=pgbouncer
Group=pgbouncer
Environment=BOUNCERCONF=/opt/tantor/etc/pgbouncer/pgbouncer.ini
ExecStart=/opt/tantor/usr/bin/pgbouncer -q $BOUNCERCONF
ExecReload=/bin/kill -HUP $MAINPID
TimeoutSec=300
LimitNOFILE=infinity
LimitNOFILESoft=infinity
[Install]
WantedBy=multi-user.target
```

и перезапустить службу:

```
systemctl daemon-reload && systemctl restart pgbouncer-tantor
```

Установка значения отличного от `infinity` заставляет ядро собирать статистику открытых файлов, что не ускоряет его работу. Лимиты увеличатся:

```
for PID in $(pgrep "pgbouncer"); do cat /proc/$PID/limits | grep files; done
Max open files 1048576  1048576  files
```

# Аутентификация клиентов: MD5

- PgBouncer самостоятельно аутентифицирует клиентов
- PgBouncer аутентифицирует клиентов способами, указанными в параметре `auth_mode`:
  - **md5** (по умолчанию) - пароли в файле `auth_file` могут указываться во всех форматах

```
set password_encryption = md5;
create user bob password 'bob';
WARNING: setting an MD5-encrypted password
DETAIL: MD5 password support is deprecated and will be removed
select rolname, rolpassword from pg_authid where rolname='bob';
 rolname |             rolpassword
---------+-------------------------------------
 bob     | md5e8557d12f6551b2ddd26bbdd0395465c
select 'md5' || md5('bob' || 'bob') md5;
                md5
-------------------------------------
 md5e8557d12f6551b2ddd26bbdd0395465c
```



## Аутентификация клиентов: MD5

PgBouncer самостоятельно аутентифицирует клиентов, которые подсоединяются на его порт и имеет собственный список имён пользователей в текстовом файле, путь к которому задаётся параметром `auth_file`.

Если имени пользователя в файле `auth_file` нет, то можно установить параметр `auth_user`, указав имя пользователя из файла `auth_file` и, тогда PgBouncer подключится к под пользователем `auth_user` к базе данных, чей описатель указан в параметре `auth_dbname` и выполнит запрос из параметра `auth_query`, получив хэш пароля пользователя, под которым хочет подсоединиться клиент. Список описателей указывается в секции `[databases]`.

Если хэш пароля нужно забрать из нескольких кластеров баз данных, то параметры `auth_user, auth_dbname, auth_query` можно переопределить, указав их (необязательно все указывать) в строке описателя базы данных, к которой хочет подключиться пользователь.

Значение по умолчанию для `auth_dbname` = **`SELECT rolname, CASE WHEN rolvaliduntil < now() THEN NULL ELSE rolpassword END FROM pg_authid WHERE rolname=$1 AND rolcanlogin`**

Выполнение запроса вносит задержку в установление соединения.

В запрос можно добавить проверку `pg_is_in_recovery()`, если нужно подсоединяться к мастеру или реплике, но при продвижении реплики до мастера, существующие соединения не закрываются и лучше использовать другие решения, например, HAproxy+Patroni.

**PgBouncer аутентифицирует клиентов способами, указанными в параметре `auth_mode`:**

md5 (по умолчанию) - в файле `auth_file` пароли могут указываться во всех форматах: текстом, в формате MD5, SCRAM. Для хэшей в формате SCRAM аутентификация будет выполняться по протоколу SCRAM, для хэшей MD5 - по протоколу MD5. Для текста - может SCRAM или MD5.

Формула создания хэша пароля типа MD5 в PostgreSQL и PgBouncer:

**`select 'md5' || md5('password' || 'username');`**

Чтобы предупреждения о "deprecated" не выдавались можно установить параметр PostgreSQL: **`alter system set md5_password_warnings = off;`**

# Аутентификация клиентов: SCRAM

- scram-sha-256 - пароли в файле должны быть указаны текстом или в формате SCRAM, формат MD5 не работает
- Пароли SCRAM могут использоваться для подключения к базам PostgreSQL, только если:
  - клиент аутентифицируется в PgBouncer по SCRAM
  - строка хэша SCRAM в PostgreSQL и в `auth_file` одинаковы

Пароль в формате SCRAM d файле `userlist.txt`:

```
"alice" "SCRAM-SHA-256$4096:pmo2r2hEB80jHCgcrw7mgQ==$...
```

правило кластера PostgreSQL в файле $PGDATA/`pg_hba.conf`:

```
host    all    all    127.0.0.1/32    scram-sha-256
```

Адрес базы в секции `[databases]` файла `pgbouncer.ini`:

```
db01 = host=127.0.0.1 port=5432 dbname=postgres user=postgres
```

Подсоединение после ввода пароля успшно:

```
psql -p 6432 -h localhost -U alice -d db01
Password for user alice: alice
db01=# select user;
 postgres
```



## Аутентификация клиентов: SCRAM

scram-sha-256 - пароли в файле должны быть указаны текстом или хэшем SCRAM. Проверка выполняется только методом SCRAM. Паролями в формате MD5 нельзя аутентифицироваться.

Пароли в формате SCRAM могут использоваться для подключения к базам PostgreSQL, только если:

1) клиент аутентифицируется в PgBouncer по протоколу SCRAM

2) хэш SCRAM в PostgreSQL и в `auth_file` полностью идентичны (совпадает вся строка: пароль, соль, число итераций).

С PgBouncer поставляется скрипт **`mkauth.py`**, который выгружает хэши паролей в текстовый файл, но скрипт написан на питоне, который может не быть установлен на сервере. Для установки питона надо выполнить:

```
apt install python3 python3-psycopg2
```

Пример команды выгрузки хэшей, файл будет перезатёрт без предупреждения:

```
./mkauth.py file.txt "port=5432 user=postgres password=postgres"
```

Периодически запуская этот скрипт, можно обновлять хэши паролей пользователей.

# Другие способы аутентификации клиентов

- **ldap** - аутентификация LDAP-сервером
- **trust** - пароль не проверяется, но имя пользователя должно быть в файле `auth_file`
- **hba** - метод аутентификации клиента берётся из файла, указанного в параметре `auth_hba_file`
  - Формат файла похож на `pg_hba.conf`
  - Способы аутентификации те же, плюс **peer** и **reject**
  - Для **peer** и **cert** есть сопоставление имён
- Параметр `application_name_add_host=1` (по умолчанию, отключён) добавляет IP адрес и порт клиента к значению параметра PostgreSQL `application_name`



## Другие способы аутентификации клиентов

**trust** - пароль не проверяется, но имя пользователя должно быть в файле `auth_file`.

**ldap** - (начиная с версии 1.25, ноябрь 2025) аутентификация LDAP-сервером, параметры задаются в `auth_ldap_options`:

```
auth_ldap_options=ldapurl="ldap://127.0.0.1:389/dc=dba1,dc=ru?uid?sub"
```

Задержка на обращение к LDAP при частых соединениях нивелирует преимущество пулера.

**hba** - метод аутентификации клиента берётся из файла, указанного в параметре `auth_hba_file`. Позволяет задать разные методы аутентификации в зависимости от параметров соединения. Формат файла похож на `pg_hba.conf`, с несущественными ограничениями. Способы аутентификации те же самые плюс **peer** и **reject**, kerberos не поддерживается. Для **peer** и **cert** поддерживается сопоставление имён (опция `map=`) с помощью файла, имя которого задаётся параметром `auth_ident_file`.

Также есть методы: **cert** (по полю CN в сертификате клиента), **plain, any, pam**, но они не практичны.

Способа аутентификации **gss (kerberos) нет**.

Параметр `application_name_add_host=1` (по умолчанию 0, отключён) добавляет IP адрес и порт клиента к значению параметра PostgreSQL `application_name`. Это может быть полезно для сопоставления источника запросов с записями в диагностическом журнале PostgreSQL и аудита. После получения соединения из пула, клиент может заменить значение параметра `application_name` и PgBouncer не будет вносить изменения в значение. У PgBouncer есть параметры `log_connections`, `log_disconnections`, включённые по умолчанию.

# Консоль управления PgBouncer

- соединиться с базой pgbouncer
- можно выполнять перечисленные команды
- право подсоединения есть у пользователей в параметрах admin_users и stats_users

```
psql -p 6432 -d pgbouncer
pgbouncer=# show help;
NOTICE:  Console usage
    SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS
    SHOW PEERS|PEER_POOLS|SERVERS|USERS|VERSION
    SHOW FDS|SOCKETS|ACTIVE_SOCKETS|LISTS
    SHOW DNS_HOSTS|DNS_ZONES|MEM|STATE
    SHOW STATS|STATS_TOTALS|STATS_AVERAGES|TOTALS
    SET key = arg
    RELOAD
    PAUSE [<db>]
    RESUME [<db>]
    DISABLE <db>
    ENABLE <db>
    RECONNECT [<db>]
    KILL [<db>]
    KILL_CLIENT <client_id>
    SUSPEND
    SHUTDOWN
    SHUTDOWN WAIT_FOR_SERVERS|WAIT_FOR_CLIENTS
    WAIT_CLOSE [<db>]
```



## Консоль управления PgBouncer

Для управления (Admin console) PgBouncer используется подключение к базе данных с именем pgbouncer. Подключение устанавливается к процессу pgbouncer. В этом подключении можно давать команды управления процессом pgbouncer.

Право на управления имеют пользователи, перечисленные в параметре `admin_users`, а мониторинга - `stats_users`.

Команда `show help;` показывает список команд:

`SHOW HELP| CONFIG | DATABASES | POOLS | CLIENTS | SERVERS | USERS | VERSION`

`SHOW PEERS | PEER_POOLS`

`SHOW FDS | SOCKETS | ACTIVE_SOCKETS | LISTS | MEM | STATE`

`SHOW DNS_HOSTS | DNS_ZONES`

`SHOW STATS | STATS_TOTALS | STATS_AVERAGES | TOTALS`

`SET key = arg` - меняет значения параметра конфигурации

`RELOAD` - переречитывание файлов параметров, посылает сигнал `HUP`

`PAUSE [<db>]` - закрывает сессии пула по мере возвращения в пул. Не вернёт промпт, пока все прежние сессии не будут завершены. Новые соединения висят и ждут команды `RESUME`.

`RESUME [<db>]` - возвращает обслуживание клиентов после `PAUSE`, `SUSPEND`, `KILL`

`DISABLE <db>`

`ENABLE <db>`

`RECONNECT [<db>]`

`KILL [<db>]` - **немедленно** завершить клиентские и серверные соединения с базой или со всеми базами, кроме псевдобазы `pgbouncer`. Новые соединения ждут команды `RESUME`.

`KILL_CLIENT <client_id>` - отсоединить клиента

`SUSPEND` - приостановка прослушивания в сокетах, таймаут 10 секунд установлен параметром `suspend_timeout`

`SHUTDOWN` - посылает сигнал **немедленной** остановки `SIGQUIT`

`SHUTDOWN WAIT_FOR_SERVERS` - посылает сигнал `SIGINT`, поведение аналогично `PAUSE` (но без прослушивания новых соединений) и `SHUTDOWN`

`SHUTDOWN WAIT_FOR_CLIENTS` - посылает сигнал `SIGTERM`, ждёт возврата сессий в пулы

`WAIT_CLOSE [<db>]` - Не вернёт промпт, пока соединения, начатые до команд `RECONNECT` или `RELOAD` не будут завершены. Используют, чтобы дождаться завершения всех старых сессий.

# Тестирование производительности

- у Unix-сокета задержка меньше, чем у TCP
- PgBench может принимать соединения по TCP и подсоединяться к PostgreSQL через Unix-сокет
- очистка сессии `DISCARD ALL` не вносит задержку
- Примеры значений tps теста pgbench:
  - через Unix-сокет к PostgreSQL: `900`
  - через localhost к PostgreSQL: `770`
  - через localhost к PgBouncer, затем Unix-сокет: `500`
  - транзакция в новой сессии к PostgreSQL: `120`
  - транзакция в новой сессии через PgBouncer: `380`



## Тестирование производительности

PgBouncer добавляет задержку. Для оценки задержки можно использовать `pgbench`.

Соединение по через Unix-сокет к экземпляру:

```
pgbench -i
pgbench -p 5432 -T 20
```

tps 900

Соединение по TCP к экземпляру:

```
pgbench -h localhost -p 5432 -T 20
```

tps 770

PgBouncer соединяется с PostgreSQL через Unix-сокет, а `pgbench` соединяется через TCP. Такая конфигурация типична, так как сервера приложений, обычно, работают на хостах, отличных от хоста, где запущен экземпляр PostgreSQL и одно соединение (от клиентов до пулера или от пулера до экземпляра) сетевое:

```
pgbench -h localhost -p 6432 -T 20
```

tps 500

Соединение через PgBouncer создаёт задержку, из-за этого tps в 1,5 раза меньше (как в режиме **session**, так и **transaction**), так как задержка добавляется к каждой SQL-команде, посылаемой в сессии.

Посмотрим то, для чего используется PgBouncer: если приложение часто создаёт сессии. Добавим параметр `-C`, чтобы `pgbench` создавал сессии для каждой транзакции. При соединении напрямую к экземпляру, серверные процессы часто запускаются и останавливаются, поэтому, tps существенно снизился:

```
pgbench -h localhost -p 5432 -T 20 -C
```

tps 120

При соединении через PgBouncer:

```
pgbench -h localhost -p 6432 -T 20 -C
```

tps 380 - при подсоединении PgBouncer к PostgreSQL по Unix-сокет

tps 350 - при подсоединении PgBouncer к PostgreSQL по 127.0.0.1

При соединении через PgBench tps больше 3 раза - серверные процессы используются повторно. Установка `server_reset_query_always=1` не влияет на tps, что означает, что команда очистки состояния сессии `DISCARD ALL` не даёт накладных расходов.

# Перезапуск процесса pgbouncer

- выполняется при обновлении версии PgBouncer или при изменении параметров, которые не могут быть применены без пересоздания сессий
- Процедура:
  - запустить новый процесс **pgbouncer** на том же порту с параметром `so_reuserport=1` и нужными изменениями
  - Послать сигнал на остановку прежнему процессу:

```
psql -h /var/run/pgbouncer-tantor/2 -p 6433 -d pgbouncer -c
"SHUTDOWN WAIT_FOR_CLIENTS"
```

  - или

```
kill -SIGTERM `head -1 /var/run/pgbouncer-tantor/pgbouncer2.pid`
```

  - если прежний процесс не остановился за приемлемое время, послать ему сигнал `SIGQUIT`



## Перезапуск процесса pgbouncer

Перезапуск процесса pgbouncer может потребоваться при обновлении его версии или при изменении параметров, которые не могут быть применены без пересоздания сессий. Для пулов **session** разрыв сессии pgbouncer с серверным процессом приводит к разрыву сессии pgbouncer с клиентом.

Параметр `-R` (`--reboot`) исполняемого файла pgbouncer объявлен deprecated.

Рекомендуется запускать новый процесс на том же порту (`so_reuserport=1`) и останавливать прежний процесс, чтобы прежние сессии доработали на старом процессе. Процедура называется "**rolling restart**" и описана в разделе документации к команде `SHUTDOWN WAIT_FOR_CLIENTS`. Процессу pgbouncer передаётся сигнал `SIGTERM`, по которому он перестаёт принимать новые соединения, ждёт отсоединения всех старых клиентов. Если дать сигнал второй раз, то это будет эквивалентно `SIGQUIT` (немедленная остановка).

Процедура:

1) запустить новый процесс **pgbouncer** на том же порту с параметром `so_reuserport=1` и новыми значениями параметров, которые нужно применить или даже новую версию pgbouncer.

2) Послать сигнал на остановку прежнему процессу:

**psql -h *unix_socket_dir* -p *listen_port* -d pgbouncer -c "SHUTDOWN WAIT_FOR_CLIENTS"**

или

**kill -SIGTERM `head -1 *pidfile*`**

3) подождать, когда прежний процесс остановится (удалит *pidfile*). Если процесс не остановился за приемлемое время, можно послать сигнал `SIGQUIT`.

Остановка службы командой:

**systemctl restart pgbouncer-tantor**

отправляет сигнал `SIGTERM`. Команда **systemctl** вернёт промпт сразу не остановив процесс, а процесс может висеть до 5 минут (параметр `TimeoutSec` в файле описателя службы), после чего процесс будет остановлен сигналом `SIGKILL`. Если процесс остановится до истечения таймаута, то процессу будет послан сигнал `SIGCONT`.