--- title: "pgBouncer - A lightweight connection pooler for PostgreSQL" draft: false ---

pgBouncer - A Lightweight Connection Pooler for PostgreSQL

PgBouncer

Lightweight connection pooler for PostgreSQL.

Homepage: https://www.pgbouncer.org/

Sources, bug tracking: https://github.com/pgbouncer/pgbouncer

Building

PgBouncer depends on few things to get compiled:

- GNU Make 3.81+
- Libevent 2.0+
- pkg-config
- OpenSSL 1.0.1+ for TLS support
- (optional) c-ares as alternative to Libevent's evdns
- (optional) PAM libraries

When dependencies are installed just run:

```
$ ./configure --prefix=/usr/local
$ make
$ make install
```

If you are building from Git, or are building for Windows, please see separate build instructions below.

DNS lookup support

PgBouncer does host name lookups at connect time instead of just once at configuration load time. This requires an asynchronous DNS implementation. The following table shows supported backends and their probing order:

backend	parallel	EDNS0 (1)	/etc/hosts	SOA lookup (2)	note
c-ares	yes	yes	yes	yes	IPv6+CNAME buggy in
evdns, libevent 2.x	yes	no	yes	no	does not check /etc/hosts
getaddrinfo_a, glibc 2.9+	yes	yes (3)	yes	no	N/A on non-glibc
getaddrinfo, libc	no	yes (3)	yes	no	requires pthreads

1. EDNS0 is required to have more than 8 addresses behind one host name.

2. SOA lookup is needed to re-check host names on zone serial change.
3. To enable EDNS0, add `options edns0` to `/etc/resolv.conf`.

c-ares is the most fully-featured implementation and is recommended for most uses and binary packaging (if a sufficiently new version is available). Libevent's built-in evdns is also suitable for many uses, with the listed restrictions. The other backends are mostly legacy options at this point and don't receive much testing anymore.

By default, c-ares is used if it can be found. Its use can be forced with `configure --with-cares` or disabled with `--without-cares`. If c-ares is not used (not found or disabled), then Libevent is used. Specify `--disable-evdns` to disable the use of Libevent's evdns and fall back to a libc-based implementation.

PAM authentication

To enable PAM authentication, `./configure` has a flag `--with-pam` (default value is no). When compiled with PAM support, a new global authentication type `pam` is available to validate users through PAM.

systemd integration

To enable systemd integration, use the `configure` option `--with-systemd`. This allows using `Type=notify` service units as well as socket activation. See `etc/pgbouncer.service` and `etc/pgbouncer.socket` for examples.

Building from Git

Building PgBouncer from Git requires that you fetch the libusual submodule and generate the header and configuration files before you can run `configure`:

```
$ git clone https://github.com/pgbouncer/pgbouncer.git
$ cd pgbouncer
$ git submodule init
$ git submodule update
$ ./autogen.sh
$ ./configure ...
$ make
$ make install
```

Additional packages required: autoconf, automake, libtool, pandoc

Testing

See the `README.md` file in the test directory on how to run the tests.

Building on Windows

The only supported build environment on Windows is MinGW. Cygwin and Visual $ANYTHING are not supported.

To build on MinGW, do the usual:

```
$ ./configure ...
$ make
```

If cross-compiling from Unix:

```
$ ./configure --host=i586-mingw32msvc ...
```

Running on Windows

Running from the command line goes as usual, except that the `-d` (daemonize), `-R` (reboot), and `-u` (switch user) switches will not work.

To run PgBouncer as a Windows service, you need to configure the `service_name` parameter to set a name for the service. Then:

```
$ pgbouncer -regservice config.ini
```

To uninstall the service:

```
$ pgbouncer -unregservice config.ini
```

To use the Windows event log, set `syslog = 1` in the configuration file. But before that, you need to register `pgbevent.dll`:

```
$ regsvr32 pgbevent.dll
```

To unregister it, do:

```
$ regsvr32 /u pgbevent.dll
```

--- title: "Usage" draft: false ---

pgbouncer

Synopsis

```
pgbouncer [-d][-R][-v][-u user] <pgbouncer.ini>
pgbouncer -V|-h
```

On Windows, the options are:

```
pgbouncer.exe [-v][-u user] <pgbouncer.ini>
pgbouncer.exe -V|-h
```

Additional options for setting up a Windows service:

```
pgbouncer.exe --regservice   <pgbouncer.ini>
pgbouncer.exe --unregservice <pgbouncer.ini>
```

Description

pgbouncer is a PostgreSQL connection pooler. Any target application can be connected to **pgbouncer** as if it were a PostgreSQL server, and **pgbouncer** will create a connection to the actual server, or it will reuse one of its existing connections.

The aim of **pgbouncer** is to lower the performance impact of opening new connections to PostgreSQL.

In order not to compromise transaction semantics for connection pooling, **pgbouncer** supports several types of pooling when rotating connections:

Session pooling Most polite method. When a client connects, a server connection will be assigned to it for the whole duration the client stays connected. When the client disconnects, the server connection will be put back into the pool. This is the default method.

Transaction pooling A server connection is assigned to a client only during a transaction. When PgBouncer notices that transaction is over, the server connection will be put back into the pool.

Statement pooling Most aggressive method. The server connection will be put back into the pool immediately after a query completes. Multi-statement transactions are disallowed in this mode as they would break.

The administration interface of **pgbouncer** consists of some new `SHOW` commands available when connected to a special "virtual" database **pgbouncer**.

Quick-start

Basic setup and usage is as follows.

1. Create a pgbouncer.ini file. Details in **pgbouncer(5)**. Simple example:

   ```
   [databases]
   template1 = host=localhost port=5432 dbname=template1

   [pgbouncer]
   listen_port = 6432
   listen_addr = localhost
   auth_type = md5
   auth_file = userlist.txt
   logfile = pgbouncer.log
   pidfile = pgbouncer.pid
   admin_users = someuser
   ```

2. Create a `userlist.txt` file that contains the users allowed in:

   ```
   "someuser" "same_password_as_in_server"
   ```

3. Launch **pgbouncer**:

   ```
   $ pgbouncer -d pgbouncer.ini
   ```

4. Have your application (or the **psql** client) connect to **pgbouncer** instead of directly to the PostgreSQL server:

   ```
   $ psql -p 6432 -U someuser template1
   ```

5. Manage **pgbouncer** by connecting to the special administration database **pgbouncer** and issuing `SHOW HELP;` to begin:

   ```
   $ psql -p 6432 -U someuser pgbouncer
   pgbouncer=# SHOW HELP;
   NOTICE:  Console usage
   DETAIL:
     SHOW [HELP|CONFIG|DATABASES|FDS|POOLS|CLIENTS|SERVERS|SOCKETS|LISTS|VERSION|...]
     SET key = arg
     RELOAD
     PAUSE
     SUSPEND
     RESUME
     SHUTDOWN
     [...]
   ```

6. If you made changes to the pgbouncer.ini file, you can reload it with:

   ```
   pgbouncer=# RELOAD;
   ```

Command line switches

`-d, --daemon` Run in the background. Without it, the process will run in the foreground.

In daemon mode, setting `pidfile` as well as `logfile` or `syslog` is required. No log messages will be written to stderr after going into the background.

Note: Does not work on Windows; **pgbouncer** need to run as service there.

`-R, --reboot` DEPRECATED: Instead of this option use a rolling restart with multiple pgbouncer processes listening on the same port using so_reuseport instead Do an online restart. That means connecting to the running process, loading the open sockets from it, and then using them. If there is no active process, boot normally. Note: Works only if OS supports Unix sockets and the `unix_socket_dir` is not disabled in configuration. Does not work on Windows. Does not work with TLS connections, they are dropped.

-u *USERNAME*, --user=*USERNAME* Switch to the given user on startup.

-v, --verbose Increase verbosity. Can be used multiple times.

-q, --quiet Be quiet: do not log to stderr. This does not affect logging verbosity, only that stderr is not to be used. For use in init.d scripts.

-V, --version Show version.

-h, --help Show short help.

--regservice Win32: Register pgbouncer to run as Windows service. The **service_name** configuration parameter value is used as the name to register under.

--unregservice Win32: Unregister Windows service.

Admin console

The console is available by connecting as normal to the database **pgbouncer**:

```
$ psql -p 6432 pgbouncer
```

Only users listed in the configuration parameters **admin_users** or **stats_users** are allowed to log in to the console. (Except when `auth_type=any`, then any user is allowed in as a stats_user.)

Additionally, the user name **pgbouncer** is allowed to log in without password, if the login comes via the Unix socket and the client has same Unix user UID as the running process.

The admin console currently only supports the simple query protocol. Some drivers use the extended query protocol for all commands; these drivers will not work for this.

Show commands

The **SHOW** commands output information. Each command is described below.

SHOW STATS Shows statistics. In this and related commands, the total figures are since process start, the averages are updated every `stats_period`.

database Statistics are presented per database.
total_xact_count Total number of SQL transactions pooled by **pgbouncer**.
total_query_count Total number of SQL commands pooled by **pgbouncer**.
total_received Total volume in bytes of network traffic received by **pgbouncer**.
total_sent Total volume in bytes of network traffic sent by **pgbouncer**.

total_xact_time Total number of microseconds spent by **pgbouncer** when connected to PostgreSQL in a transaction, either idle in transaction or executing queries.

total_query_time Total number of microseconds spent by **pgbouncer** when actively connected to PostgreSQL, executing queries.

total_wait_time Time spent by clients waiting for a server, in microseconds. Updated when a client connection is assigned a backend connection.

avg_xact_count Average transactions per second in last stat period.

avg_query_count Average queries per second in last stat period.

avg_recv Average received (from clients) bytes per second.

avg_sent Average sent (to clients) bytes per second.

avg_xact_time Average transaction duration, in microseconds.

avg_query_time Average query duration, in microseconds.

avg_wait_time Average time spent by clients waiting for a server that were assigned a backend connection within the current `stats_period`, in microseconds (averaged per second within that period).

SHOW STATS_TOTALS Subset of **SHOW STATS** showing the total values (**total_**).

SHOW STATS_AVERAGES Subset of **SHOW STATS** showing the average values (**avg_**).

SHOW TOTALS Like **SHOW STATS** but aggregated across all databases.

SHOW SERVERS

type S, for server.

user User name **pgbouncer** uses to connect to server.

database Database name.

state State of the pgbouncer server connection, one of **active**, **idle**, **used**, **tested**, **new**, **active_cancel**, **being_canceled**.

addr IP address of PostgreSQL server.

port Port of PostgreSQL server.

local_addr Connection start address on local machine.

local_port Connection start port on local machine.

connect_time When the connection was made.

request_time When last request was issued.

wait Not used for server connections.

wait_us Not used for server connections.

close_needed 1 if the connection will be closed as soon as possible, because a configuration file reload or DNS update changed the connection information or **RECONNECT** was issued.

ptr Address of internal object for this connection. Used as unique ID.

link Address of client connection the server is paired with.

remote_pid PID of backend server process. In case connection is made over Unix socket and OS supports getting process ID info, its OS PID. Otherwise it's extracted from cancel packet the server sent, which should be the PID in case the server is PostgreSQL, but it's a random number in case the server it is another PgBouncer.

tls A string with TLS connection information, or empty if not using TLS.

application_name A string containing the `application_name` set on the linked client connection, or empty if this is not set, or if there is no linked connection.

prepared_statements The amount of prepared statements that are prepared on the server. This number is limited by the `max_prepared_statements` setting.

SHOW CLIENTS

type C, for client.

user Client connected user.

database Database name.

state State of the client connection, one of **active**, **waiting**, **active_cancel_req**, or **waiting_cancel_req**.

addr IP address of client.

port Source port of client.

local_addr Connection end address on local machine.

local_port Connection end port on local machine.

connect_time Timestamp of connect time.

request_time Timestamp of latest client request.

wait Current waiting time in seconds.

wait_us Microsecond part of the current waiting time.

close_needed not used for clients

ptr Address of internal object for this connection. Used as unique ID.

link Address of server connection the client is paired with.

remote_pid Process ID, in case client connects over Unix socket and OS supports getting it.

tls A string with TLS connection information, or empty if not using TLS.

application_name A string containing the `application_name` set by the client for this connection, or empty if this was not set.

prepared_statements The amount of prepared statements that the client has prepared

SHOW POOLS A new pool entry is made for each couple of (database, user).

database Database name.

user User name.

cl_active Client connections that are either linked to server connections or are idle with no queries waiting to be processed.

cl_waiting Client connections that have sent queries but have not yet got a

server connection.

cl_active_cancel_req Client connections that have forwarded query cancellations to the server and are waiting for the server response.

cl_waiting_cancel_req Client connections that have not forwarded query cancellations to the server yet.

sv_active Server connections that are linked to a client.

sv_active_cancel Server connections that are currently forwarding a cancel request.

sv_being_canceled Servers that normally could become idle but are waiting to do so until all in-flight cancel requests have completed that were sent to cancel a query on this server.

sv_idle Server connections that are unused and immediately usable for client queries.

sv_used Server connections that have been idle for more than `server_check_delay`, so they need `server_check_query` to run on them before they can be used again.

sv_tested Server connections that are currently running either `server_reset_query` or `server_check_query`.

sv_login Server connections currently in the process of logging in.

maxwait How long the first (oldest) client in the queue has waited, in seconds. If this starts increasing, then the current pool of servers does not handle requests quickly enough. The reason may be either an overloaded server or just too small of a **pool_size** setting.

maxwait_us Microsecond part of the maximum waiting time.

pool_mode The pooling mode in use.

SHOW PEER_POOLS A new peer_pool entry is made for each configured peer.

database ID of the configured peer entry.

cl_active_cancel_req Client connections that have forwarded query cancellations to the server and are waiting for the server response.

cl_waiting_cancel_req Client connections that have not forwarded query cancellations to the server yet.

sv_active_cancel Server connections that are currently forwarding a cancel request.

sv_login Server connections currently in the process of logging in.

SHOW LISTS Show following internal information, in columns (not rows):

databases Count of databases.

users Count of users.

pools Count of pools.

free_clients Count of free clients. These are clients that are disconnected, but PgBouncer keeps the memory around that was allocated for them so it can be reused for a future clients to avoid allocations.

used_clients Count of used clients.
login_clients Count of clients in **login** state.
free_servers Count of free servers. These are servers that are disconnected, but PgBouncer keeps the memory around that was allocated for them so it can be reused for a future servers to avoid allocations.
used_servers Count of used servers.
dns_names Count of DNS names in the cache.
dns_zones Count of DNS zones in the cache.
dns_queries Count of in-flight DNS queries.
dns_pending not used

SHOW USERS

name The user name
pool_mode The user's override pool_mode, or NULL if the default will be used instead.

SHOW DATABASES

name Name of configured database entry.
host Host pgbouncer connects to.
port Port pgbouncer connects to.
database Actual database name pgbouncer connects to.
force_user When the user is part of the connection string, the connection between pgbouncer and PostgreSQL is forced to the given user, whatever the client user.
pool_size Maximum number of server connections.
min_pool_size Minimum number of server connections.
reserve_pool Maximum number of additional connections for this database.
pool_mode The database's override pool_mode, or NULL if the default will be used instead.
max_connections Maximum number of allowed connections for this database, as set by **max_db_connections**, either globally or per database.
current_connections Current number of connections for this database.
paused 1 if this database is currently paused, else 0.
disabled 1 if this database is currently disabled, else 0.

SHOW PEERS

peer_id ID of the configured peer entry.
host Host pgbouncer connects to.
port Port pgbouncer connects to.
pool_size Maximum number of server connections that can be made to this peer

SHOW FDS Internal command - shows list of file descriptors in use with internal state attached to them.

When the connected user has the user name "pgbouncer", connects through the Unix socket and has same the UID as the running process, the actual FDs are passed over the connection. This mechanism is used to do an online restart. Note: This does not work on Windows.

This command also blocks the internal event loop, so it should not be used while PgBouncer is in use.

fd File descriptor numeric value.
task One of **pooler**, **client** or **server**.
user User of the connection using the FD.
database Database of the connection using the FD.
addr IP address of the connection using the FD, **unix** if a Unix socket is used.
port Port used by the connection using the FD.
cancel Cancel key for this connection.
link fd for corresponding server/client. NULL if idle.

SHOW SOCKETS, SHOW ACTIVE_SOCKETS Shows low-level information about sockets or only active sockets. This includes the information shown under **SHOW CLIENTS** and **SHOW SERVERS** as well as other more low-level information.

SHOW CONFIG Show the current configuration settings, one per row, with the following columns:

key Configuration variable name
value Configuration value
default Configuration default value
changeable Either **yes** or **no**, shows if the variable can be changed while running. If **no**, the variable can be changed only at boot time. Use **SET** to change a variable at run time.

SHOW MEM Shows low-level information about the current sizes of various internal memory allocations. The information presented is subject to change.

SHOW DNS_HOSTS Show host names in DNS cache.

hostname Host name.
ttl How many seconds until next lookup.
addrs Comma separated list of addresses.

SHOW DNS_ZONES Show DNS zones in cache.

zonename Zone name.
serial Current serial.
count Host names belonging to this zone.

SHOW VERSION Show the PgBouncer version string.

SHOW STATE Show the PgBouncer state settings. Current states are active, paused and suspended.

Process controlling commands

PAUSE [db] PgBouncer tries to disconnect from all servers. Disconnecting each server connection waits for that server connection to be released according to the server pool's pooling mode (in transaction pooling mode, the transaction must complete, in statement mode, the statement must complete, and in session pooling mode the client must disconnect). The command will not return before all server connections have been disconnected. To be used at the time of database restart.

If database name is given, only that database will be paused.

New client connections to a paused database will wait until **RESUME** is called.

DISABLE db Reject all new client connections on the given database.

ENABLE db Allow new client connections after a previous **DISABLE** command.

RECONNECT [db] Close each open server connection for the given database, or all databases, after it is released (according to the pooling mode), even if its lifetime is not up yet. New server connections can be made immediately and will connect as necessary according to the pool size settings.

This command is useful when the server connection setup has changed, for example to perform a gradual switchover to a new server. It is *not* necessary to run this command when the connection string in pgbouncer.ini has been changed and reloaded (see **RELOAD**) or when DNS resolution has changed, because then the equivalent of this command will be run automatically. This command is only necessary if something downstream of PgBouncer routes the connections.

After this command is run, there could be an extended period where some server connections go to an old destination and some server connections go to a new destination. This is likely only sensible when switching read-only traffic between read-only replicas, or when switching between nodes of a multimaster replication setup. If all connections need to be switched at the same time, **PAUSE** is recommended instead. To close server connections without waiting (for example, in emergency failover rather than gradual switchover scenarios), also consider **KILL**.

KILL db Immediately drop all client and server connections on given database.

New client connections to a killed database will wait until **RESUME** is called.

SUSPEND All socket buffers are flushed and PgBouncer stops listening for data on them. The command will not return before all buffers are empty. To be used at the time of PgBouncer online reboot.

New client connections to a suspended database will wait until **RESUME** is called.

RESUME [db] Resume work from previous **KILL**, **PAUSE**, or **SUSPEND** command.

SHUTDOWN The PgBouncer process will exit.

RELOAD The PgBouncer process will reload its configuration files and update changeable settings. This includes the main configuration file as well as the files specified by the settings `auth_file` and `auth_hba_file`.

PgBouncer notices when a configuration file reload changes the connection parameters of a database definition. An existing server connection to the old destination will be closed when the server connection is next released (according to the pooling mode), and new server connections will immediately use the updated connection parameters.

WAIT_CLOSE [db] Wait until all server connections, either of the specified database or of all databases, have cleared the "close_needed" state (see **SHOW SERVERS**). This can be called after a **RECONNECT** or **RELOAD** to wait until the respective configuration change has been fully activated, for example in switchover scripts.

Other commands

SET key = arg Changes a configuration setting (see also **SHOW CONFIG**). For example:

```
SET log_connections = 1;
SET server_check_query = 'select 2';
```

(Note that this command is run on the PgBouncer admin console and sets PgBouncer settings. A **SET** command run on another database will be passed to the PostgreSQL backend like any other SQL command.)

Signals

SIGHUP Reload config. Same as issuing the command **RELOAD** on the console.

SIGINT Safe shutdown. Same as issuing **PAUSE** and **SHUTDOWN** on the console.

SIGTERM Immediate shutdown. Same as issuing **SHUTDOWN** on the console.

SIGUSR1 Same as issuing **PAUSE** on the console.
SIGUSR2 Same as issuing **RESUME** on the console.

Libevent settings

From the Libevent documentation:

> It is possible to disable support for epoll, kqueue, devpoll, poll or select by setting the environment variable EVENT_NOEPOLL, EVENT_NOKQUEUE, EVENT_NODEVPOLL, EVENT_NOPOLL or EVENT_NOSELECT, respectively.
>
> By setting the environment variable EVENT_SHOW_METHOD, libevent displays the kernel notification method that it uses.

See also

pgbouncer(5) - man page of configuration settings descriptions

https://www.pgbouncer.org/

--- title: "Configuration" draft: false ---

pgbouncer.ini

Description

The configuration file is in "ini" format. Section names are between "[" and "]". Lines starting with ";" or "#" are taken as comments and ignored. The characters ";" and "#" are not recognized as special when they appear later in the line.

Generic settings

logfile

Specifies the log file. For daemonization (`-d`), either this or `syslog` need to be set.

The log file is kept open, so after rotation, `kill -HUP` or on console `RELOAD;` should be done. On Windows, the service must be stopped and started.

Note that setting `logfile` does not by itself turn off logging to stderr. Use the command-line option `-q` or `-d` for that.

Default: not set

pidfile

Specifies the PID file. Without `pidfile` set, daemonization (`-d`) is not allowed.

Default: not set

listen_addr

Specifies a list (comma-separated) of addresses where to listen for TCP connections. You may also use `*` meaning "listen on all addresses". When not set, only Unix socket connections are accepted.

Addresses can be specified numerically (IPv4/IPv6) or by name.

Default: not set

listen_port

Which port to listen on. Applies to both TCP and Unix sockets.

Default: 6432

unix_socket_dir

Specifies the location for Unix sockets. Applies to both the listening socket and to server connections. If set to an empty string, Unix sockets are disabled. A value that starts with `@` specifies that a Unix socket in the abstract namespace should be created (currently supported on Linux and Windows).

For online reboot (`-R`) to work, a Unix socket needs to be configured, and it needs to be in the file-system namespace.

Default: `/tmp` (empty on Windows)

unix_socket_mode

File system mode for Unix socket. Ignored for sockets in the abstract namespace. Not supported on Windows.

Default: 0777

unix_socket_group

Group name to use for Unix socket. Ignored for sockets in the abstract namespace. Not supported on Windows.

Default: not set

user

If set, specifies the Unix user to change to after startup. Works only if PgBouncer is started as root or if it's already running as the given user. Not supported on Windows.

Default: not set

pool_mode

Specifies when a server connection can be reused by other clients.

session Server is released back to pool after client disconnects. Default.
transaction Server is released back to pool after transaction finishes.
statement Server is released back to pool after query finishes. Transactions spanning multiple statements are disallowed in this mode.

max_client_conn

Maximum number of client connections allowed.

When this setting is increased, then the file descriptor limits in the operating system might also have to be increased. Note that the number of file descriptors potentially used is more than `max_client_conn`. If each user connects under its own user name to the server, the theoretical maximum used is:

```
max_client_conn + (max pool_size * total databases * total users)
```

If a database user is specified in the connection string (all users connect under the same user name), the theoretical maximum is:

```
max_client_conn + (max pool_size * total databases)
```

The theoretical maximum should never be reached, unless somebody deliberately crafts a special load for it. Still, it means you should set the number of file descriptors to a safely high number.

Search for `ulimit` in your favorite shell man page. Note: `ulimit` does not apply in a Windows environment.

Default: 100

default_pool_size

How many server connections to allow per user/database pair. Can be overridden in the per-database configuration.

Default: 20

min_pool_size

Add more server connections to pool if below this number. Improves behavior when the normal load suddenly comes back after a period of total inactivity. The value is effectively capped at the pool size.

Only enforced for pools where at least one of the following is true:

- the entry in the `[database]` section for the pool has a value set for the `user` key (aka forced user)
- there is at least one client connected to the pool

Default: 0 (disabled)

reserve_pool_size

How many additional connections to allow to a pool (see `reserve_pool_timeout`). 0 disables.

Default: 0 (disabled)

reserve_pool_timeout

If a client has not been serviced in this time, use additional connections from the reserve pool. 0 disables. [seconds]

Default: 5.0

max_db_connections

Do not allow more than this many server connections per database (regardless of user). This considers the PgBouncer database that the client has connected to, not the PostgreSQL database of the outgoing connection.

This can also be set per database in the `[databases]` section.

Note that when you hit the limit, closing a client connection to one pool will not immediately allow a server connection to be established for another pool, because the server connection for the first pool is still open. Once the server connection closes (due to idle timeout), a new server connection will immediately be opened for the waiting pool.

Default: 0 (unlimited)

max_user_connections

Do not allow more than this many server connections per user (regardless of database). This considers the PgBouncer user that is associated with a pool, which is either the user specified for the server connection or in absence of that the user the client has connected as.

This can also be set per user in the `[users]` section.

Note that when you hit the limit, closing a client connection to one pool will not immediately allow a server connection to be established for another pool, because the server connection for the first pool is still open. Once the server connection closes (due to idle timeout), a new server connection will immediately be opened for the waiting pool.

Default: 0 (unlimited)

server_round_robin

By default, PgBouncer reuses server connections in LIFO (last-in, first-out) manner, so that few connections get the most load. This gives best performance if you have a single server serving a database. But if there is a round-robin system behind a database address (TCP, DNS, or host list), then it is better if PgBouncer also uses connections in that manner, thus achieving uniform load.

Default: 0

track_extra_parameters

By default, PgBouncer tracks `client_encoding`, `datestyle`, `timezone`, `standard_conforming_strings` and `application_name` parameters per client. To allow other parameters to be tracked, they can be specified here, so that PgBouncer knows that they should be maintained in the client variable cache and restored in the server whenever the client becomes active.

If you need to specify multiple values, use a comma-separated list (e.g. `default_transaction_readonly, IntervalStyle`)

Note: Most parameters cannot be tracked this way. The only parameters that can be tracked are ones that Postgres reports to the client. Postgres has an official list of parameters that it reports to the client. Postgres extensions can change this list though, they can add parameters themselves that they also report, and they can start reporting already existing parameters that Postgres does not report. Notably Citus 12.0+ causes Postgres to also report `search_path`.

The Postgres protocol allows specifying parameters settings, both directly as a parameter in the startup packet, or inside the `options` startup packet. Parameters specified using both of these methods are supported by `track_extra_parameters`. However, it's not possible to include `options` itself in `track_extra_parameters`, only the parameters contained in `options`.

Default: IntervalStyle

ignore_startup_parameters

By default, PgBouncer allows only parameters it can keep track of in startup packets: `client_encoding`, `datestyle`, `timezone` and `standard_conforming_strings`. All others parameters will raise an error. To allow others parameters, they can

be specified here, so that PgBouncer knows that they are handled by the admin and it can ignore them.

If you need to specify multiple values, use a comma-separated list (e.g. `options,extra_float_digits`)

The Postgres protocol allows specifying parameters settings, both directly as a parameter in the startup packet, or inside the `options` startup packet. Parameters specified using both of these methods are supported by `ignore_startup_parameters`. It's even possible to include `options` itself in `track_extra_parameters`, which results in any unknown parameters contained inside `options` to be ignored.

Default: empty

peer_id

The peer id used to identify this PgBouncer process in a group of PgBouncer processes that are peered together. The `peer_id` value should be unique within a group of peered PgBouncer processes. When set to 0 pgbouncer peering is disabled. See the docs for the `[peers]` section for more information. The maximum value that can be used for the `peer_id` is 16383.

Default: 0

disable_pqexec

Disable the Simple Query protocol (PQexec). Unlike the Extended Query protocol, Simple Query allows multiple queries in one packet, which allows some classes of SQL-injection attacks. Disabling it can improve security. Obviously, this means only clients that exclusively use the Extended Query protocol will stay working.

Default: 0

application_name_add_host

Add the client host address and port to the application name setting set on connection start. This helps in identifying the source of bad queries etc. This logic applies only at the start of a connection. If `application_name` is later changed with `SET`, PgBouncer does not change it again.

Default: 0

conffile

Show location of current config file. Changing it will make PgBouncer use another config file for next `RELOAD` / `SIGHUP`.

Default: file from command line

service_name

Used on win32 service registration.

Default: `pgbouncer`

job_name

Alias for `service_name`.

stats_period

Sets how often the averages shown in various `SHOW` commands are updated and how often aggregated statistics are written to the log (but see `log_stats`). [seconds]

Default: 60

max_prepared_statements

When this is set to a non-zero value PgBouncer tracks protocol-level named prepared statements related commands sent by the client in transaction and statement pooling mode. PgBouncer makes sure that any statement prepared by a client is available on the backing server connection. Even when the statement was originally prepared on another server connection.

PgBouncer internally examines all the queries that are sent as a prepared statement by clients and gives each unique query string an internal name with the format `PGBOUNCER_{unique_id}`. If the same query string is prepared multiple times (possibly by different clients), then these queries share the same internal name. PgBouncer only prepares the statement on the actual PostgreSQL server using the internal name (so not the name provided by the client). PgBouncer keeps track of the name that the client gave to each prepared statement. It then rewrites each command that uses a prepared statement to by replacing the client side name with the the internal name (e.g. replacing `my_prepared_stamenent` with `PGBOUNCER_123`) before forwarding that command to the server. More importantly, if the prepared statement that the client wants to execute is not yet prepared on the server (e.g. because a different server is now assigned to the client then when the client prepared the statement), then PgBouncer transparrently prepares the statement before executing it.

Note: This tracking and rewriting of prepared statement commands does not work for SQL-level prepared statement commands, so `PREPARE`, `EXECUTE` and `DEALLOCATE` are forwarded straight to Postgres. The exception to this rule are the `DEALLOCATE ALL` and `DISCARD ALL` commands, these do work as expected and will clear the prepared statements that PgBouncer tracked for the client that sends this command.

The actual value of this setting controls the number of prepared statements kept active in an LRU cache on a single server connection. When the setting

is set to 0 prepared statement support for transaction and statement pooling is disabled. To get the best performance you should try to make sure that this setting is larger than the amount of commonly used prepared statements in your application. Keep in mind that the higher this value, the larger the memory footprint of each PgBouncer connection will be on your PostgreSQL server, because it will keep more queries prepared on those connections. It also increases the memory footprint of PgBouncer itself, because it now needs to keep track of query strings.

The impact on PgBouncer memory usage is not that big though: - Each unique query is stored once in a global query cache. - Each client connection keeps a buffer that it uses to rewrite packets. This is at most 4 times the size of `pkt_buf`. This limit is often not reached though, it only happens when the queries in your prepared statements are between 2 and 4 times the size of `pkt_buf`.

So if you consider the following as an example scenario: - There are 1000 active clients - The clients prepare 200 unique queries - The average size of a query is 5kB - `pkt_buf` parameter is set to the default of 4096 (4kB)

Then PgBouncer needs at most the following amount of memory to handle these prepared statements:

200 x 5kB + 1000 x 4 x 4kB = ~17MB of memory.

Tracking prepared statements does not only come with a memory cost, but also with increased CPU usage, because PgBouncer needs to inspect and rewrite the queries. Multiple PgBouncer instances can listen on the same port to use more than one core for processing, see the documentation for the `so_reuseport` option for details.

But of course there are also performance benefits to prepared statements. Just as when connecting to PostgreSQL directly, by preparing a query that is executed many times, it reduces the total amount of parsing and planning that needs to be done. The way that PgBouncer tracks prepared statements is especially beneficial to performance when multiple clients prepare the same queries. Because client connections automatically reuse a prepared statement on a server connection even if it was prepared by another client. As an example if you have a `pool_size` of 20 and you have 100 clients that all prepare the exact same query, then the query is prepared (and thus parsed) only 20 times on the PostgreSQL server.

The reuse of prepared statements has one downside. If the return or argument types of a prepared statement changes across executions then PostgreSQL currently throws an error such as:

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

You can avoid such errors by not having multiple clients that use the exact same query string in a prepared statement, but expecting different argument or result types. One of the most common ways of running into this issue is during a DDL migration where you add a new column or change a column type on an existing

table. In those cases you can run `RECONNECT` on the PgBouncer admin console after doing the migration to force a re-prepare of the query and make the error goes away.

Default: 0

Authentication settings

PgBouncer handles its own client authentication and has its own database of users. These settings control this.

auth_type

How to authenticate users.

cert Client must connect over TLS connection with a valid client certificate. The user name is then taken from the CommonName field from the certificate.

md5 Use MD5-based password check. This is the default authentication method. `auth_file` may contain both MD5-encrypted and plain-text passwords. If `md5` is configured and a user has a SCRAM secret, then SCRAM authentication is used automatically instead.

scram-sha-256 Use password check with SCRAM-SHA-256. `auth_file` has to contain SCRAM secrets or plain-text passwords.

plain The clear-text password is sent over the wire. Deprecated.

trust No authentication is done. The user name must still exist in `auth_file`.

any Like the `trust` method, but the user name given is ignored. Requires that all databases are configured to log in as a specific user. Additionally, the console database allows any user to log in as admin.

hba The actual authentication type is loaded from `auth_hba_file`. This allows different authentication methods for different access paths, for example: connections over Unix socket use the `peer` auth method, connections over TCP must use TLS.

pam PAM is used to authenticate users, `auth_file` is ignored. This method is not compatible with databases using the `auth_user` option. The service name reported to PAM is "pgbouncer". `pam` is not supported in the HBA configuration file.

auth_hba_file

HBA configuration file to use when `auth_type` is `hba`.

Default: not set

auth_file

The name of the file to load user names and passwords from. See section Authentication file format below about details.

Most authentication types (see above) require that either `auth_file` or `auth_user` be set; otherwise there would be no users defined.

Default: not set

auth_user

If `auth_user` is set, then any user not specified in `auth_file` will be queried through the `auth_query` query from pg_shadow in the database, using `auth_user`. The password of `auth_user` will be taken from `auth_file`. (If the `auth_user` does not require a password then it does not need to be defined in `auth_file`.)

Direct access to pg_shadow requires admin rights. It's preferable to use a non-superuser that calls a SECURITY DEFINER function instead.

Default: not set

auth_query

Query to load user's password from database.

Direct access to pg_shadow requires admin rights. It's preferable to use a non-superuser that calls a SECURITY DEFINER function instead.

Note that the query is run inside the target database. So if a function is used, it needs to be installed into each database.

Default: `SELECT usename, passwd FROM pg_shadow WHERE usename=$1`

auth_dbname

Database name in the `[database]` section to be used for authentication purposes. This option can be either global or overridden in the connection string if this parameter is specified.

Log settings

syslog

Toggles syslog on/off. On Windows, the event log is used instead.

Default: 0

syslog_ident

Under what name to send logs to syslog.

Default: `pgbouncer` (program name)

syslog_facility

Under what facility to send logs to syslog. Possibilities: `auth`, `authpriv`, `daemon`, `user`, `local0-7`.

Default: `daemon`

log_connections

Log successful logins.

Default: 1

log_disconnections

Log disconnections with reasons.

Default: 1

log_pooler_errors

Log error messages the pooler sends to clients.

Default: 1

log_stats

Write aggregated statistics into the log, every `stats_period`. This can be disabled if external monitoring tools are used to grab the same data from `SHOW` commands.

Default: 1

verbose

Increase verbosity. Mirrors the "-v" switch on the command line. For example, using "-v -v" on the command line is the same as `verbose=2`.

Default: 0

Console access control

admin_users

Comma-separated list of database users that are allowed to connect and run all commands on the console. Ignored when `auth_type` is `any`, in which case any user name is allowed in as admin.

Default: empty

stats_users

Comma-separated list of database users that are allowed to connect and run read-only queries on the console. That means all `SHOW` commands except `SHOW FDS`.

Default: empty

Connection sanity checks, timeouts

server_reset_query

Query sent to server on connection release, before making it available to other clients. At that moment no transaction is in progress, so the value should not include `ABORT` or `ROLLBACK`.

The query is supposed to clean any changes made to the database session so that the next client gets the connection in a well-defined state. The default is `DISCARD ALL`, which cleans everything, but that leaves the next client no pre-cached state. It can be made lighter, e.g. `DEALLOCATE ALL` to just drop prepared statements, if the application does not break when some state is kept around.

When transaction pooling is used, the `server_reset_query` is not used, because in that mode, clients must not use any session-based features, since each transaction ends up in a different connection and thus gets a different session state.

Default: `DISCARD ALL`

server_reset_query_always

Whether `server_reset_query` should be run in all pooling modes. When this setting is off (default), the `server_reset_query` will be run only in pools that are in sessions-pooling mode. Connections in transaction-pooling mode should not have any need for a reset query.

This setting is for working around broken setups that run applications that use session features over a transaction-pooled PgBouncer. It changes non-deterministic breakage to deterministic breakage: Clients always lose their state after each transaction.

Default: 0

server_check_delay

How long to keep released connections available for immediate re-use, without running `server_check_query` on it. If 0 then the check is always run.

Default: 30.0

server_check_query

Simple do-nothing query to check if the server connection is alive.

If an empty string, then sanity checking is disabled.

Default: `select 1`

server_fast_close

Disconnect a server in session pooling mode immediately or after the end of the current transaction if it is in "close_needed" mode (set by `RECONNECT`, `RELOAD` that changes connection settings, or DNS change), rather than waiting for the session end. In statement or transaction pooling mode, this has no effect since that is the default behavior there.

If because of this setting a server connection is closed before the end of the client session, the client connection is also closed. This ensures that the client notices that the session has been interrupted.

This setting makes connection configuration changes take effect sooner if session pooling and long-running sessions are used. The downside is that client sessions are liable to be interrupted by a configuration change, so client applications will need logic to reconnect and reestablish session state. But note that no transactions will be lost, because running transactions are not interrupted, only idle sessions.

Default: 0

server_lifetime

The pooler will close an unused (not currently linked to any client connection) server connection that has been connected longer than this. Setting it to 0 means the connection is to be used only once, then closed. [seconds]

Default: 3600.0

server_idle_timeout

If a server connection has been idle more than this many seconds it will be closed. If 0 then this timeout is disabled. [seconds]

Default: 600.0

server_connect_timeout

If connection and login don't finish in this amount of time, the connection will be closed. [seconds]

Default: 15.0

server_login_retry

If login to the server failed, because of failure to connect or from authentication, the pooler waits this much before retrying to connect. During the waiting interval, new clients trying to connect to the failing server will get an error immediately without another connection attempt. [seconds]

The purpose of this behavior is that clients don't unnecessarily queue up waiting for a server connection to become available if the server is not working. However, it also means that if a server is momentarily failing, for example during a restart or if the configuration was erroneous, then it will take at least this long until the pooler will consider connecting to it again. Planned events such as restarts should normally be managed using the `PAUSE` command to avoid this.

Default: 15.0

client_login_timeout

If a client connects but does not manage to log in in this amount of time, it will be disconnected. Mainly needed to avoid dead connections stalling `SUSPEND` and thus online restart. [seconds]

Default: 60.0

autodb_idle_timeout

If the automatically created (via "*") database pools have been unused this many seconds, they are freed. The negative aspect of that is that their statistics are also forgotten. [seconds]

Default: 3600.0

dns_max_ttl

How long DNS lookups can be cached. The actual DNS TTL is ignored. [seconds]

Default: 15.0

dns_nxdomain_ttl

How long DNS errors and NXDOMAIN DNS lookups can be cached. [seconds]

Default: 15.0

dns_zone_check_period

Period to check if a zone serial has changed.

PgBouncer can collect DNS zones from host names (everything after first dot) and then periodically check if the zone serial changes. If it notices changes, all

host names under that zone are looked up again. If any host IP changes, its connections are invalidated.

Works only with c-ares backend (`configure` option `--with-cares`).

Default: 0.0 (disabled)

resolv_conf

The location of a custom `resolv.conf` file. This is to allow specifying custom DNS servers and perhaps other name resolution options, independent of the global operating system configuration.

Requires evdns (>= 2.0.3) or c-ares (>= 1.15.0) backend.

The parsing of the file is done by the DNS backend library, not PgBouncer, so see the library's documentation for details on allowed syntax and directives.

Default: empty (use operating system defaults)

TLS settings

client_tls_sslmode

TLS mode to use for connections from clients. TLS connections are disabled by default. When enabled, `client_tls_key_file` and `client_tls_cert_file` must be also configured to set up the key and certificate PgBouncer uses to accept client connections.

disable Plain TCP. If client requests TLS, it's ignored. Default.
allow If client requests TLS, it is used. If not, plain TCP is used. If the client presents a client certificate, it is not validated.
prefer Same as `allow`.
require Client must use TLS. If not, the client connection is rejected. If the client presents a client certificate, it is not validated.
verify-ca Client must use TLS with valid client certificate.
verify-full Same as `verify-ca`.

client_tls_key_file

Private key for PgBouncer to accept client connections.

Default: not set

client_tls_cert_file

Certificate for private key. Clients can validate it.

Default: not set

client_tls_ca_file

Root certificate file to validate client certificates.

Default: not set

client_tls_protocols

Which TLS protocol versions are allowed. Allowed values: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Shortcuts: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (all).

Default: `secure`

client_tls_ciphers

Allowed TLS ciphers, in OpenSSL syntax. Shortcuts:

- `default/secure/fast/normal` (these all use system wide OpenSSL defaults)
- `all` (enables all ciphers, not recommended)

Only connections using TLS version 1.2 and lower are affected. There is currently no setting that controls the cipher choices used by TLS version 1.3 connections.

Default: `default`

client_tls_ecdhcurve

Elliptic Curve name to use for ECDH key exchanges.

Allowed values: `none` (DH is disabled), `auto` (256-bit ECDH), curve name

Default: `auto`

client_tls_dheparams

DHE key exchange type.

Allowed values: `none` (DH is disabled), `auto` (2048-bit DH), `legacy` (1024-bit DH)

Default: `auto`

server_tls_sslmode

TLS mode to use for connections to PostgreSQL servers. The default mode is `prefer`.

disable Plain TCP. TLS is not even requested from the server.
allow FIXME: if server rejects plain, try TLS?

prefer TLS connection is always requested first from PostgreSQL. If refused, the connection will be established over plain TCP. Server certificate is not validated. Default

require Connection must go over TLS. If server rejects it, plain TCP is not attempted. Server certificate is not validated.

verify-ca Connection must go over TLS and server certificate must be valid according to `server_tls_ca_file`. Server host name is not checked against certificate.

verify-full Connection must go over TLS and server certificate must be valid according to `server_tls_ca_file`. Server host name must match certificate information.

server_tls_ca_file

Root certificate file to validate PostgreSQL server certificates.

Default: not set

server_tls_key_file

Private key for PgBouncer to authenticate against PostgreSQL server.

Default: not set

server_tls_cert_file

Certificate for private key. PostgreSQL server can validate it.

Default: not set

server_tls_protocols

Which TLS protocol versions are allowed. Allowed values: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Shortcuts: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (all).

Default: `secure`

server_tls_ciphers

Allowed TLS ciphers, in OpenSSL syntax. Shortcuts:

- `default/secure/fast/normal` (these all use system wide OpenSSL defaults)
- `all` (enables all ciphers, not recommended)

Only connections using TLS version 1.2 and lower are affected. There is currently no setting that controls the cipher choices used by TLS version 1.3 connections.

Default: `default`

Dangerous timeouts

Setting the following timeouts can cause unexpected errors.

query_timeout

Queries running longer than that are canceled. This should be used only with a slightly smaller server-side `statement_timeout`, to apply only for network problems. [seconds]

Default: 0.0 (disabled)

query_wait_timeout

Maximum time queries are allowed to spend waiting for execution. If the query is not assigned to a server during that time, the client is disconnected. 0 disables. If this is disabled, clients will be queued indefinitely. [seconds]

This setting is used to prevent unresponsive servers from grabbing up connections. It also helps when the server is down or rejects connections for any reason.

Default: 120.0

cancel_wait_timeout

Maximum time cancellation requests are allowed to spend waiting for execution. If the cancel request is not assigned to a server during that time, the client is disconnected. 0 disables. If this is disabled, cancel requests will be queued indefinitely. [seconds]

This setting is used to prevent a client locking up when a cancel cannot be forwarded due to the server being down.

Default: 10.0

client_idle_timeout

Client connections idling longer than this many seconds are closed. This should be larger than the client-side connection lifetime settings, and only used for network problems. [seconds]

Default: 0.0 (disabled)

idle_transaction_timeout

If a client has been in "idle in transaction" state longer, it will be disconnected. [seconds]

Default: 0.0 (disabled)

suspend_timeout

How long to wait for buffer flush during `SUSPEND` or reboot (`-R`). A connection is dropped if the flush does not succeed. [seconds]

Default: 10

Low-level network settings

pkt_buf

Internal buffer size for packets. Affects size of TCP packets sent and general memory usage. Actual libpq packets can be larger than this, so no need to set it large.

Default: 4096

max_packet_size

Maximum size for PostgreSQL packets that PgBouncer allows through. One packet is either one query or one result set row. The full result set can be larger.

Default: 2147483647

listen_backlog

Backlog argument for listen(2). Determines how many new unanswered connection attempts are kept in the queue. When the queue is full, further new connections are dropped.

Default: 128

sbuf_loopcnt

How many times to process data on one connection, before proceeding. Without this limit, one connection with a big result set can stall PgBouncer for a long time. One loop processes one `pkt_buf` amount of data. 0 means no limit.

Default: 5

so_reuseport

Specifies whether to set the socket option `SO_REUSEPORT` on TCP listening sockets. On some operating systems, this allows running multiple PgBouncer instances on the same host listening on the same port and having the kernel distribute the connections automatically. This option is a way to get PgBouncer to use more CPU cores. (PgBouncer is single-threaded and uses one CPU core per instance.)

The behavior in detail depends on the operating system kernel. As of this writing, this setting has the desired effect on (sufficiently recent versions of)

Linux, DragonFlyBSD, and FreeBSD. (On FreeBSD, it applies the socket option `SO_REUSEPORT_LB` instead.) Some other operating systems support the socket option but it won't have the desired effect: It will allow multiple processes to bind to the same port but only one of them will get the connections. See your operating system's setsockopt() documentation for details.

On systems that don't support the socket option at all, turning this setting on will result in an error.

Each PgBouncer instance on the same host needs different settings for at least `unix_socket_dir` and `pidfile`, as well as `logfile` if that is used. Also note that if you make use of this option, you can no longer connect to a specific PgBouncer instance via TCP/IP, which might have implications for monitoring and metrics collection.

To make sure query cancellations keep working, you should set up PgBouncer peering between the different PgBouncer processes. For details look at docs for the `peer_id` configuration option and the `peers` configuration section. There's also an example that uses peering and so_reuseport in the example section of these docs.

Default: 0

tcp_defer_accept

Sets the `TCP_DEFER_ACCEPT` socket option; see `man 7 tcp` for details. (This is a Boolean option: 1 means enabled. The actual value set if enabled is currently hardcoded to 45 seconds.)

This is currently only supported on Linux.

Default: 1 on Linux, otherwise 0

tcp_socket_buffer

Default: not set

tcp_keepalive

Turns on basic keepalive with OS defaults.

On Linux, the system defaults are tcp_keepidle=7200, tcp_keepintvl=75, tcp_keepcnt=9. They are probably similar on other operating systems.

Default: 1

tcp_keepcnt

Default: not set

tcp_keepidle

Default: not set

tcp_keepintvl

Default: not set

tcp_user_timeout

Sets the `TCP_USER_TIMEOUT` socket option. This specifies the maximum amount of time in milliseconds that transmitted data may remain unacknowledged before the TCP connection is forcibly closed. If set to 0, then operating system's default is used.

This is currently only supported on Linux.

Default: 0

Section [databases]

The section `[databases]` defines the names of the databases that clients of PgBouncer can connect to and specifies where those connections will be routed. The section contains key=value lines like

```
dbname = connection string
```

where the key will be taken as a database name and the value as a connection string, consisting of key=value pairs of connection parameters, described below (similar to libpq, but the actual libpq is not used and the set of available features is different). Example:

```
foodb = host=host1.example.com port=5432
bardb = host=localhost dbname=bazdb
```

The database name can contain characters `_0-9A-Za-z` without quoting. Names that contain other characters need to be quoted with standard SQL identifier quoting: double quotes, with "" for a single instance of a double quote.

The database name "pgbouncer" is reserved for the admin console and cannot be used as a key here.

"*" acts as a fallback database: If the exact name does not exist, its value is taken as connection string for the requested database. For example, if there is an entry (and no other overriding entries)

```
* = host=foo
```

then a connection to PgBouncer specifying a database "bar" will effectively behave as if an entry

```
bar = host=foo dbname=bar
```

exists (taking advantage of the default for `dbname` being the client-side database name; see below).

Such automatically created database entries are cleaned up if they stay idle longer than the time specified by the `autodb_idle_timeout` parameter.

dbname

Destination database name.

Default: same as client-side database name

host

Host name or IP address to connect to. Host names are resolved at connection time, the result is cached per `dns_max_ttl` parameter. When a host name's resolution changes, existing server connections are automatically closed when they are released (according to the pooling mode), and new server connections immediately use the new resolution. If DNS returns several results, they are used in a round-robin manner.

If the value begins with `/`, then a Unix socket in the file-system namespace is used. If the value begins with `@`, then a Unix socket in the abstract namespace is used.

A comma-separated list of host names or addresses can be specified. In that case, connections are made in a round-robin manner. (If a host list contains host names that in turn resolve via DNS to multiple addresses, the round-robin systems operate independently. This is an implementation dependency that is subject to change.) Note that in a list, all hosts must be available at all times: There are no mechanisms to skip unreachable hosts or to select only available hosts from a list or similar. (This is different from what a host list in libpq means.) Also note that this only affects how the destinations of new connections are chosen. See also the setting `server_round_robin` for how clients are assigned to already established server connections.

Examples:

```
host=localhost
host=127.0.0.1
host=2001:0db8:85a3:0000:0000:8a2e:0370:7334
host=/var/run/postgresql
host=192.168.0.1,192.168.0.2,192.168.0.3
```

Default: not set, meaning to use a Unix socket

port

Default: 5432

user

If `user=` is set, all connections to the destination database will be done with the specified user, meaning that there will be only one pool for this database.

Otherwise, PgBouncer logs into the destination database with the client user name, meaning that there will be one pool per user.

password

If no password is specified here, the password from the `auth_file` or `auth_query` will be used.

auth_user

Override of the global `auth_user` setting, if specified.

auth_query

Override of the global `auth_query` setting, if specified. The entire SQL statement needs to be enclosed in single quotes.

auth_dbname

Override of the global `auth_dbname` setting, if specified.

pool_size

Set the maximum size of pools for this database. If not set, the `default_pool_size` is used.

min_pool_size

Set the minimum pool size for this database. If not set, the global `min_pool_size` is used.

Only enforced if at least one of the following is true:

- this entry in the `[database]` section has a value set for the `user` key (aka forced user)
- there is at least one client connected to the pool

reserve_pool

Set additional connections for this database. If not set, `reserve_pool_size` is used.

connect_query

Query to be executed after a connection is established, but before allowing the connection to be used by any clients. If the query raises errors, they are logged but ignored otherwise.

pool_mode

Set the pool mode specific to this database. If not set, the default `pool_mode` is used.

max_db_connections

Configure a database-wide maximum (i.e. all pools within the database will not have more than this many server connections).

client_encoding

Ask specific `client_encoding` from server.

datestyle

Ask specific `datestyle` from server.

timezone

Ask specific `timezone` from server.

Section [users]

This section contains key=value lines like

```
user1 = settings
```

where the key will be taken as a user name and the value as a list of key=value pairs of configuration settings specific for this user. Example:

```
user1 = pool_mode=session
```

Only a few settings are available here.

pool_mode

Set the pool mode to be used for all connections from this user. If not set, the database or default `pool_mode` is used.

max_user_connections

Configure a maximum for the user (i.e. all pools with the user will not have more than this many server connections).

Section [peers]

The section `[peers]` defines the peers that PgBouncer can forward cancellation requests to and where those cancellation requests will be routed.

PgBouncer processes can be peered together in a group by defining a `peer_id` value and a `[peers]` section in the configs of all the PgBouncer processes. These PgBouncer processes can then forward cancellations requests to the process that it originated from. This is needed to make cancellations work when multiple PgBouncer processes (possibly on different servers) are behind the same TCP load balancer. Cancellation requests are sent over different TCP connections than the query they are cancelling, so a TCP load balancer might send the cancellation request connection to a different process than the one that it was meant for. By peering them these cancellation requests eventually end up at the right process. A more in-depth explanation is provided in this recording of a conference talk.

The section contains key=value lines like

```
peer_id = connection string
```

Where the key will be taken as a `peer_id` and the value as a connection string, consisting of key=value pairs of connection parameters, described below (similar to libpq, but the actual libpq is not used and the set of available features is different). Example:

```
1 = host=host1.example.com
2 = host=/tmp/pgbouncer-2  port=5555
```

Note 1: For peering to work, the `peer_id` of each PgBouncer process in the group must be unique within the peered group. And the `[peers]` section should contain entries for each of those peer ids. An example can be found in the examples section of these docs. It **is** allowed, but not necessary, for the `[peers]` section to contain the `peer_id` of the PgBouncer that the config is for. Such an entry will be ignored, but it is allowed to config management easy. Because it allows using the exact same `[peers]` section for multiple configs.

Note 2: Cross-version peering is supported as long as all peers are on the same side of the v1.21.0 version boundary. In v1.21.0 some breaking changes were made in how we encode the cancellation tokens that made them incompatible with the ones created by earlier versions.

host

Host name or IP address to connect to. Host names are resolved at connection time, the result is cached per `dns_max_ttl` parameter. If DNS returns several results, they are used in a round-robin manner. But in general it's not recommended to use a hostname that resolves to multiple IPs, because then the cancel request might still be forwarded to the wrong node and it would need to be forwarded again (which is only allowed up to three times).

If the value begins with `/`, then a Unix socket in the file-system namespace is used. If the value begins with `@`, then a Unix socket in the abstract namespace is used.

Examples:

```
host=localhost
host=127.0.0.1
host=2001:0db8:85a3:0000:0000:8a2e:0370:7334
host=/var/run/pgbouncer-1
```

port

Default: 6432

pool_size

Set the maximum number of cancel requests that can be in flight to the peer at the same time. It's quite normal for cancel requests to arrive in bursts, e.g. when the backing Postgres server slow or down. So it's important for `pool_size` to not be so low that it cannot handle these bursts.

If not set, the `default_pool_size` is used.

Include directive

The PgBouncer configuration file can contain include directives, which specify another configuration file to read and process. This allows splitting the configuration file into physically separate parts. The include directives look like this:

```
%include filename
```

If the file name is not an absolute path, it is taken as relative to the current working directory.

Authentication file format

This section describes the format of the file specified by the `auth_file` setting. It is a text file in the following format:

```
"username1" "password" ...
"username2" "md5abcdef012342345" ...
"username2" "SCRAM-SHA-256$<iterations>:<salt>$<storedkey>:<serverkey>"
```

There should be at least 2 fields, surrounded by double quotes. The first field is the user name and the second is either a plain-text, a MD5-hashed password, or a SCRAM secret. PgBouncer ignores the rest of the line. Double quotes in a field value can be escaped by writing two double quotes.

PostgreSQL MD5-hashed password format:

```
"md5" + md5(password + username)
```

So user `admin` with password `1234` will have MD5-hashed password `md545f2603610af569b6155c45067268c6b`.

PostgreSQL SCRAM secret format:

```
SCRAM-SHA-256$<iterations>:<salt>$<storedkey>:<serverkey>
```

See the PostgreSQL documentation and RFC 5803 for details on this.

The passwords or secrets stored in the authentication file serve two purposes. First, they are used to verify the passwords of incoming client connections, if a password-based authentication method is configured. Second, they are used as the passwords for outgoing connections to the backend server, if the backend server requires password-based authentication (unless the password is specified directly in the database's connection string). The latter works if the password is stored in plain text or MD5-hashed. SCRAM secrets can only be used for logging into a server if the client authentication also uses SCRAM, the PgBouncer database definition does not specify a user name, and the SCRAM secrets are identical in PgBouncer and the PostgreSQL server (same salt and iterations, not merely the same password). This is due to an inherent security property of SCRAM: The stored SCRAM secret cannot by itself be used for deriving login credentials.

The authentication file can be written by hand, but it's also useful to generate it from some other list of users and passwords. See `./etc/mkauth.py` for a sample script to generate the authentication file from the `pg_shadow` system table. Alternatively, use `auth_query` instead of `auth_file` to avoid having to maintain a separate authentication file.

HBA file format

The location of the HBA file is specified by the setting `auth_hba_file`. It is only used if `auth_type` is set to `hba`.

The file follows the format of the PostgreSQL `pg_hba.conf` file (see https://www.postgresql.org/docs/current/auth-pg-hba-conf.html).

- Supported record types: `local`, `host`, `hostssl`, `hostnossl`.
- Database field: Supports `all`, `sameuser`, `@file`, multiple names. Not supported: `replication`, `samerole`, `samegroup`.
- User name field: Supports `all`, `@file`, multiple names. Not supported: `+groupname`.
- Address field: Supports IPv4, IPv6. Not supported: DNS names, domain prefixes.
- Auth-method field: Only methods supported by PgBouncer's `auth_type` are supported, plus `peer` and `reject`, but except `any` and `pam`, which only work globally. User name map (`map=`) parameter is not supported.

Examples

Small example configuration:

```
[databases]
template1 = host=localhost dbname=template1 auth_user=someuser

[pgbouncer]
pool_mode = session
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
stats_users = stat_collector
```

Database examples:

```
[databases]

; foodb over Unix socket
foodb =

; redirect bardb to bazdb on localhost
bardb = host=localhost dbname=bazdb

; access to destination database will go with single user
forcedb = host=localhost port=300 user=baz password=foo client_encoding=UNICODE datestyle=IS
```

Example of a secure function for `auth_query`:

```
CREATE OR REPLACE FUNCTION pgbouncer.user_lookup(in i_username text, out uname text, out pha
RETURNS record AS $$
BEGIN
    SELECT usename, passwd FROM pg_catalog.pg_shadow
    WHERE usename = i_username INTO uname, phash;
    RETURN;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
REVOKE ALL ON FUNCTION pgbouncer.user_lookup(text) FROM public, pgbouncer;
GRANT EXECUTE ON FUNCTION pgbouncer.user_lookup(text) TO pgbouncer;
```

Example configs for 2 peered PgBouncer processes to create a multi-core PgBouncer setup using `so_reuseport`. The config for the first process:

```
[databases]
postgres = host=localhost dbname=postgres
```

```
[peers]
1 = host=/tmp/pgbouncer1
2 = host=/tmp/pgbouncer2

[pgbouncer]
listen_addr=127.0.0.1
auth_file=auth_file.conf
so_reuseport=1
unix_socket_dir=/tmp/pgbouncer1
peer_id=1
```

The config for the second process:

```
[databases]
postgres = host=localhost dbname=postgres

[peers]
1 = host=/tmp/pgbouncer1
2 = host=/tmp/pgbouncer2

[pgbouncer]
listen_addr=127.0.0.1
auth_file=auth_file.conf
so_reuseport=1
; only unix_socket_dir and peer_id are different
unix_socket_dir=/tmp/pgbouncer2
peer_id=2
```

See also

pgbouncer(1) - man page for general usage, console commands

https://www.pgbouncer.org/

--- title: PgBouncer TODO list draft: false ---

PgBouncer TODO list

Highly visible missing features

Significant amount of users feel the need for those.

- Protocol-level plan cache.
- LISTEN/NOTIFY. Requires strict SQL format.

Waiting for contributors...

Problems / cleanups

- Bad naming in data strctures:
  - PgSocket->db [vs. PgPool->db]
- other per-user settings
- Maintenance order vs. lifetime_kill_gap: http://lists.pgfoundry.org/pipermail/pgbouncer-general/2011-February/000679.html
- per_loop_maint/per_loop_activate take too much time in case of moderate load and lots of pools. Perhaps active_pool_list would help, which contains only pools touched in current loop.
- new states for clients: idle and in-query. That allows to apply client_idle_timeout and query_timeout without walking all clients on maintenance time.
- check if SQL error codes are correct
- removing user should work - kill connections
- keep stats about error counts
- cleanup of logging levels, to make log more useful
- to test:
  - signal flood
  - no mem / no fds handling
- fix high-freq maintenance timer - it's only needed when PAUSE/RESUME/shutdown is issued.
- Get rid of SBUF_SMALL_PKT logic - it makes processing code complex. Needs a new sbuf_prepare_*() to notify sbuf about short data. [Plain 'false' from handler postpones processing to next event loop.]
- units for config parameters.

Dubious/complicated features

- Load-balancing / failover. Both are already solved via DNS. Adding load-balancing config in pgbouncer might be good idea. Adding failover decision-making is not...
- User-based route. Simplest would be to move db info to pool and fill username into dns.
- some preliminary notification that fd limit is full
- Move all "look-at-full-packet" situations to SBUF_EV_PKT_CALLBACK

- `pool_mode = plproxy` - use postgres in full-duplex mode for autocommit queries, multiplexing several queries into one connection. Should result in more efficient CPU usage of server.
- SMP: spread sockets over per-cpu threads. Needs confirmation that single-threadedness can be problem. It can also be that only accept() + login handling of short connection is problem that could be solved by just having threads for login handling, which would be lot simpler or just deciding that it is not worth fixing.