Last modified: Tue Jul 23 08:04:32 JST 2013

Welcome to pgpool -II page

What is pgpool-II?

pgpool-II is a middle ware that sits between PostgreSQL servers and a PostgreSQL database client. It provides the following features:

- Connection Pooling

  pgpool-II maintains established connections to the PostgreSQL servers, and reuses them whenever a new connection with the same properties (i.e. user name, database, protocol version) comes in. It reduces the connection overhead, and improves system's overall throughput.

- Replication

  pgpool-II can manage multiple PostgreSQL servers. Activating the replication feature makes it possible to create a real time backup on 2 or more PostgreSQL clusters, so that the service can continue without interruption if one of those clusters fails.

- Load Balance

  If a database is replicated(because running in either replication mode or master/slave mode), performing a SELECT query on any server will return the same result. pgpool-II takes advantage of the replication feature in order to reduce the load on each PostgreSQL server. It does that by distributing SELECT queries among available servers, improving the system's overall throughput. In an ideal scenario, read performance could improve proportionally to the number of PostgreSQL servers. Load balancing works best in a scenario where there are a lot of users executing many read-only queries at the same time.

- Limiting Exceeding Connections

  There is a limit on the maximum number of concurrent connections with PostgreSQL, and new connections are rejected when this number is reached. Raising this maximum number of connections, however, increases resource consumption and has a negative impact on overall system performance. pgpool-II also has a limit on the maximum number of connections, but extra connections will be queued instead of returning an error immediately.

- Parallel Query

  Using the parallel query feature, data can be split among multiple servers, so that a query can be executed on all the servers concurrently, reducing the overall execution time. Parallel query works

Watchdog
PCP commands
Troubleshooting
Restrictions
internal

[Japanese page]

best when searching large-scale data.

pgpool-II speaks PostgreSQL's backend and frontend protocol, and relays messages between a backend and a frontend. Therefore, a database application (frontend) thinks that pgpool-II is the actual PostgreSQL server, and the server (backend) sees pgpool-II as one of its clients. Because pgpool-II is transparent to both the server and the client, an existing database application can be used with pgpool-II almost without a change to its source code.

There are some restrictions to using SQL via pgpool-II. See Restrictions for more details.

back to top

License

back to top

Supported Platforms

pgpool-II works on Linux, Solaris, FreeBSD, and most of the UNIX-like architectures. Windows is not supported. Supported PostgreSQL server's versions are 6.4 and higher. To use the parallel query feature, however, 7.4 and higher must be used.

If you are using PostgreSQL 7.3 or older, some features of pgpool-II won't be available. But you shouldn't use such an old release anyway.

You must also make sure that all of your PostgreSQL servers are using the same major PostgreSQL version. In addition to this, we do not recommend mixing different PostgreSQL installation with different build options: including supporting SSL or not, to use --disable-integer-datetimes or not, different block size. These might affect part of functionality of pgpool-II. The difference of PostgreSQL minor versions is not usually a problem. However we do not test

every occurrence of minor versions and we recommend to use exact same minor version of PostgreSQL.

back to top

pgpool-II Installation

pgpool-II can be downloaded from the pgpool Development page. Packages are also provided for various platforms including CentOS, RedHat Enterprise Linux, Fedora and Debian. Check appropriate repository.

pgpool-II's source code can be downloaded from: pgpool development page

Installing pgpool-II from source code requires gcc 2.9 or higher, and GNU make. Also, pgpool-II links with the libpq library, so the libpq library and its development headers must be installed on the machine used to build pgpool-II. Additionally the OpenSSL library and its development headers must be present in order to enable OpenSSL support in pgpool-II.

Installing pgpool-II

configure

After extracting the source tarball, execute the configure script.

```
./configure
```

If you want non-default values, some options can be set:

`--prefix=path`	pgpool-II binaries and docs will be installed in this directory. Default value is `/usr/local`
`--with-pgsql=path`	The top directory where PostgreSQL's client libraries are installed. Default value is provided by `pg_config`
`--with-openssl`	pgpool-II binaries will be built with OpenSSL support. OpenSSL support is disabled by default. V2.3 -
`--enable-sequence-lock`	Use insert_lock compatible with pgpool-II 3.0 series(until 3.0.4). pgpool-II locks against a row in the sequence table. PostgreSQL 8.2 or later which was released after June 2011 cannot use this lock method. V3.1 -

--enable-table-lock	Use insert_lock compatible with pgpool-II 2.2 and 2.3 series. pgpool-II locks against the insert target table. This lock method is deprecated because it causes a lock conflict with VACUUM. V3.1 -
--with-memcached=path	pgpool-II binaries will use memcached for on memory query cache. You have to install libmemcached. V3.2 -

make

```
make
make install
```

will install pgpool-II. (If you use Solaris or FreeBSD, replace make with gmake)

Installing functions

Installing pgpool_regclass (recommended) V3.0 -

If you are using PostgreSQL 8.0 or later, installing pgpool_regclass function on all PostgreSQL to be accessed by pgpool-II is strongly recommended, as it is used internally by pgpool-II. Without this, handling of duplicate table names in different schema might cause trouble (temporary tables aren't a problem).

```
$ cd pgpool-II-x.x.x/sql/pgpool-regclass
$ make
$ make install
```

After this:

```
$ psql -f pgpool-regclass.sql template1
```

or

```
$ psql template1
=# CREATE EXTENSION pgpool_regclass;
```

Executing pgpool-regclass.sql or `CREATE EXTENSION` should be performed on every databases accessed with pgpool-II. You do not need to do this for a database created after the execution of "`psql -f pgpool-regclass.sql template1`" or `CREATE`

EXTENSION, as this template database will be cloned to create new databases.

Creating insert_lock table V3.0 -

If you use insert_lock in replication mode, creating pgpool_catalog.insert_lock table for mutual exclusion is strongly recommended. Without this, insert_lock works so far. However in that case pgpool-II locks against the insert target table. This behavior is same as pgpool-II 2.2 and 2.3 series. The table lock conflicts with VACUUM. So INSERT processing may be thereby kept waiting for a long time.

```
$ cd pgpool-II-x.x.x/sql
$ psql -f insert_lock.sql template1
```

Executing insert_lock.sql should be performed on every databases accessed with pgpool-II. You do not need to do this for a database created after the execution of "`psql -f insert_lock.sql template1`", as this template database will be cloned to create new databases.

Installing pgpool_recovery

If you use online recovery, some functions are needed: pgpool_recovery, pgpool_remote_start, pgpool_switch_xlog.

And, pgpoolAdmin of the tool to control pgpool-II can stop, restart and reload the backend PostgreSQL nodes, and it needs the function named pgpool_pgctl.

Here is the way to install there functions.

```
$ cd pgpool-II-x.x.x/sql/pgpool-regclass
$ make
$ make install
$ psql -f pgpool-recovery.sql template1
```

Configuring pgpool.pgctl V3.0 -

The function pgpool_pgctl executes the value of ppool.pgctl in postgresql.conf as the pg_ctl command. To use this function, you have to this custom parameter.

```
ex)
$ cat >> /usr/local/pgsql/postgresql.conf
pgpool.pg_ctl = '/usr/local/pgsql/bin/pg_ctl'
```

```
$ pg_ctl reload -D /usr/local/pgsql/data
```

back to top

Configuring pgpool-II

Default configuration files for pgpool-II are `/usr/local/etc/pgpool.conf` and `/usr/local/etc/pcp.conf`. Several operation modes are available in pgpool-II. Each mode has associated features which can be enabled or disabled, and specific configuration parameters to control their behaviors.

Function/Mode	Raw Mode (*3)	Replication Mode	Master/Slave Mode	Parallel Query Mode
Connection Pool	X	O	O	O
Replication	X	O	X	(*1)
Load Balance	X	O	O	(*1)
Failover	O	O	O	X
Online recovery	X	0	(*2)	X
Parallel Query	X	X	X	O
Required # of Servers	1 or higher	2 or higher	2 or higher	2 or higher
System DB required?	no	no	no	yes

- O means 'available', X means 'unavailable'
- (*1) Parallel Query Mode requires the replication or the load-balancing turned on, however replication and load-balancing cannot be used for a distributed table in Parallel Query Mode.
- (*2) Online recovery can be used with Master slave+Streaming replication.
- (*3) Clients simply connect to the PostgreSQL servers via pgpool-II. This mode is useful for simply limiting excess connections to the servers, or enabling failover with multiple servers.

Configuring `pcp.conf`

pgpool-II provides a control interface where an administrator can collect pgpool-II status, and terminate pgpool-II processes remotely. `pcp.conf` is the user/password file used for authentication by this interface. All operation modes require the `pcp.conf` file to be set. A `$prefix/etc/pcp.conf.sample` file is created during the installation of pgpool-II. Rename the file to `pcp.conf` and add your user name and password to it.

```
cp $prefix/etc/pcp.conf.sample $prefix/etc/pcp.conf
```

An empty line or a line starting with "#" is treated as a comment and will be ignored. A user name and its associated password must be written as one line using the following format:

```
username:[password encrypted in md5]
```

`[password encrypted in md5]` can be produced with the `$prefix/bin/pg_md5` command.

```
pg_md5 -p
password: <your password>
```

or

```
./pg_md5 foo
acbd18db4cc2f85cedef654fccc4a4d8
```

The `pcp.conf` file must be readable by the user who executes pgpool-II.

Configuring `pgpool.conf`

As already explained, each operation mode has its specific configuration parameters in `pgpool.conf`. A `$prefix/etc/pgpool.conf.sample` file is created during the installation of pgpool-II. Rename the file to `pgpool.conf` and edit its contents.

```
cp $prefix/etc/pgpool.conf.sample $prefix/etc/pgpool.conf
```

There are additional sample pgpool.conf for each mode. V2.3 -

Mode	sample file
replication mode	pgpool.conf.sample-replication
master/slave mode(Slony-I)	pgpool.conf.sample-master-slave

master/slave mode(Streaming replication)	pgpool.conf.sample-stream

An empty line or a line starting with "#" is treated as a comment and will be ignored.

Connections

listen_addresses

Specifies the hostname or IP address, on which pgpool-II will accept TCP/IP connections. `'*'` accepts all incoming connections. `''` disables TCP/IP connections. Default is `'localhost'`. Connections via UNIX domain socket are always accepted.

This parameter can only be set at server start.

port

The port number used by pgpool-II to listen for connections. Default is 9999.

This parameter can only be set at server start.

socket_dir

The directory where the UNIX domain socket accepting connections for pgpool-II will be created. Default is `'/tmp'`. Be aware that this socket might be deleted by a cron job. We recommend to set this value to `'/var/run'` or such directory.

This parameter can only be set at server start.

pcp_port

The port number where PCP process accepts connections. Default is 9898.

This parameter can only be set at server start.

pcp_socket_dir

The directory path of the UNIX domain socket accepting connections for the PCP process. Default is `'/tmp'`. Be aware that the socket might be deleted by cron. We recommend to set this value to `'/var/run'` or such directory.

This parameter can only be set at server start.

backend_socket_dir - V3.0

> ***DEPRECATED*** : This parameter is deprecated for consistency with the default libpq policy. See the backend_hostname parameter definition to adapt your configuration accordingly.

This parameter was defining the PostgreSQL server's UNIX domain socket directory. Default is `'/tmp'`.

This parameter can only be set at server start.

Pools

num_init_children

The number of preforked pgpool-II server processes. Default is 32. num_init_children is also the concurrent connections limit to pgpool-II from clients. If more than num_init_children clients try to connect to pgpool-II, they are blocked (not rejected) until a connection to any pgpool-II process is closed. Up to 2*num_init_children can be queued. Number of connections to each PostgreSQL is roughly max_pool*num_init_children

Some hints in addition to above:

- Canceling a query creates another connection to the backend; thus, a query cannot be canceled if all the connections are in use. If you want to ensure that queries can be canceled, set this value to twice the expected connections.
- PostgreSQL allows concurrent connections for non superusers up to max_connections - superuser_reserved_connections.

In summary, max_pool, num_init_children, max_connections, superuser_reserved_connections must satisfy the following formula:

```
max_pool*num_init_children <= (max_connections - superuser_reserved_connections) (no query canceling needed)
max_pool*num_init_children*2 <= (max_connections - superuser_reserved_connections) (query canceling needed)
```

This parameter can only be set at server start.

child_life_time

A pgpool-II child process' life time in seconds. When a child is idle for that many seconds, it is terminated and a new child is created. This parameter is a measure to prevent memory leaks and other unexpected errors. Default value is 300 (5 minutes). 0 disables this feature. Note that this doesn't apply for processes that have not accepted any connection yet.

You need to reload pgpool.conf if you change this value.

child_max_connections

A pgpool-II child process will be terminated after this many connections from clients. This parameter is useful on a server if it is so busy that child_life_time and connection_life_time are never triggered. Thus this is also useful to prevent PostgreSQL servers from getting too big.

You need to reload pgpool.conf if you change this value.

client_idle_limit

Disconnect a client if it has been idle for client_idle_limit seconds after the last query has completed. This is useful to prevent pgpool childs from being occupied by a lazy client or a broken TCP/IP connection between client and pgpool. The default value for client_idle_limit is 0, which means the feature is turned off. this

value. This parameter is ignored in the second stage of online recovery.

You need to reload pgpool.conf if you change client_idle_limit.

enable_pool_hba

If true, use pool_hba.conf for client authentication. See setting up pool_hba.conf for client authentication.

You need to reload pgpool.conf if you change this value.

pool_passwd

Specify the file name of pool_passwd for md5 authentication. Default value is "pool_passwd". "" disables to read pool_passwd. See Authentication / Access Controls for more details.

You need to restart pgpool-II if you change this value.

authentication_timeout

Specify the timeout for pgpool authentication. 0 disables the time out. Default value is 60.

You need to reload pgpool.conf if you change this value.

Logs

log_destination V3.1 -

PgPool II supports several methods for logging server messages, including stderr and syslog. The default is to log to stderr.

Note: you will need to alter the configuration of your system's syslog daemon in order to make use of the syslog option for log_destination. PgPool can log to syslog facilities LOCAL0 through LOCAL7 (see syslog_facility), but the default syslog configuration on most platforms will discard all such messages. You will need to add something like

```
local0.*    /var/log/pgpool.log
```

to the syslog daemon's configuration file to make it work.

print_timestamp

Add timestamps to the logs when set to true. Default is true.

You need to reload pgpool.conf if you change print_timestamp.

log_connections

If true, all incoming connections will be printed to the log.

You need to reload pgpool.conf if you change this value.

log_hostname

If true, ps command status will show the client's hostname instead of an IP address. Also, if log_connections is enabled, hostname will be logged.

You need to reload pgpool.conf if you change this value.

log_statement

Produces SQL log messages when true. This is similar to the log_statement parameter in PostgreSQL. It produces logs even if the debug option was not passed to pgpool-II at start up.

You need to reload pgpool.conf if you change this value.

log_per_node_statement V2.3 -

Similar to log_statement, except that it prints logs for each DB node separately. It can be useful to make sure that replication is working, for example.

You need to reload pgpool.conf if you change this value.

syslog_facility V3.1 -

When logging to syslog is enabled, this parameter determines the syslog "facility" to be used. You can choose from LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7; the default is LOCAL0. See also the documentation of your system's syslog daemon.

syslog_ident V3.1 -

When logging to syslog is enabled, this parameter determines the program name used to identify PgPool messages in syslog logs. The default is pgpool.

debug_level V3.0 -

Debug message verbosity level. 0 means no message, greater than 1 means more verbose message. Default value is 0.

File locations

pid_file_name

Full path to a file which contains pgpool's process id. Default is "/var/run/pgpool/pgpool.pid".

You need to restart pgpool-II if you change this value.

logdir

pgpool_status is written into this directory.

Connection pooling

connection_cache

Caches connections to backends when set to true. Default is true.

You need to restart pgpool-II if you change this value.

Health check

health_check_timeout

pgpool-II periodically tries to connect to the backends to detect any error on the servers or networks. This error check procedure is called "health check". If an error is detected, pgpool-II tries to perform failover or degeneration.

This parameter serves to prevent the health check from waiting for a long time in a case such as unplugged network cable. The timeout value is in seconds. Default value is 20. 0 disables timeout (waits until TCP/IP timeout).

This health check requires one extra connection to each backend, so `max_connections` in the `postgresql.conf` needs to be incremented as needed.

You need to reload pgpool.conf if you change this value.

health_check_period

This parameter specifies the interval between the health checks in seconds. Default is 0, which means health check is disabled.

You need to reload pgpool.conf if you change health_check_period.

health_check_user

The user name to perform health check. This user must exist in all the PostgreSQL backends. Otherwise, health check causes an error.

You need to reload pgpool.conf if you change health_check_user.

health_check_password V3.1 -

The password of the user to perform health check.

You need to reload pgpool.conf if you change health_check_password.

health_check_max_retries V3.2 -

The maximum number of times to retry a failed health check before giving up and initiating failover. This setting can be useful in spotty networks, when it is expected that health checks will fail occasionally even when the master is fine. Default is 0, which means do not retry. It is advised that you disable fail_over_on_backend_error if you want to enable health_check_max_retries.

You need to reload pgpool.conf if you change health_check_max_retries.

health_check_retry_delay V3.2 -

The amount of time (in seconds) to sleep between failed health check retries (not used unless health_check_max_retries is > 0). If 0, then retries are immediate (no delay).

You need to reload pgpool.conf if you change health_check_retry_delay.

search_primary_node_timeout V3.3 -

The parameter specifies the maximum amount of time in seconds to search for a primary node when a failover scenario occurs. The default value for the parameter is 10. pgpool-II will search for the primary node for the amount of time given in case of failover before giving up trying to search for a primary node. 0 means keep trying forever. This parameter will be ignored if running in other than streaming replication mode.

You need to reload pgpool.conf if you change search_primary_node_timeout.

Failover and failback

failover_command

This parameter specifies a command to run when a node is detached. pgpool-II replaces the following special characters with backend specific information.

Special character	Description
%d	Backend ID of a detached node.
%h	Hostname of a detached node.
%p	Port number of a detached node.
%D	Database cluster directory of a detached node.
%M	Old master node ID.
%m	New master node ID.
%H	Hostname of the new master node.
%P	Old primary node ID.
%r	New master port number.
%R	New master database cluster directory.
%%	'%' character

You need to reload pgpool.conf if you change failover_command.

When a failover is performed, pgpool kills all its child processes, which will in turn terminate all active sessions to pgpool. Then pgpool invokes the failover_command and waits for its completion. After this, pgpool starts new child processes and is ready again to accept connections from clients.

failback_command

This parameter specifies a command to run when a node is attached. pgpool-II replaces special the following characters with backend specific information.

Special character	Description
%d	Backend ID of an attached node.
%h	Hostname of an attached node.
%p	Port number of an attached node.
%D	Database cluster path of an attached node.
%M	Old master node
%m	New master node
%H	Hostname of the new master node.
%P	Old primary node ID.
%r	New master port number.
%R	New master database cluster directory.
%%	'%' character

You need to reload pgpool.conf if you change failback_command.

follow_master_command V3.1 -

This parameter specifies a command to run in master/slave streaming replication mode only after a master failover. pgpool-II replaces the following special characters with backend specific information.

Special character	Description
%d	Backend ID of a detached node.
%h	Hostname of a detached node.
%p	Port number of a detached node.

%D	Database cluster directory of a detached node.
%M	Old master node ID.
%m	New master node ID.
%H	Hostname of the new master node.
%P	Old primary node ID.
%r	New master port number.
%R	New master database cluster directory.
%%	'%' character

You need to reload pgpool.conf if you change follow_master_command.

If follow_master_command is not empty, when a master failover is completed in master/slave streaming replication, pgpool degenerate all nodes excepted the new master and starts new child processes to be ready again to accept connections from clients. After this, pgpool run the command set into the 'follow_master_command' for each degenerated nodes. Typically the command should be used to recover the slave from the new master by call the pcp_recovery_node command for example.

fail_over_on_backend_error V2.3 -

If true, and an error occurs when reading/writing to the backend communication, pgpool-II will trigger the fail over procedure. If set to false, pgpool will report an error and disconnect the session. If you set this parameter to off, it is recommended that you turn on health checking. Please note that even if this parameter is set to off, however, pgpool will also do the fail over when pgpool detects the administrative shutdown of postmaster.

You need to reload pgpool.conf if you change this value.

Load balancing mode

ignore_leading_white_space

pgpool-II ignores white spaces at the beginning of SQL queries while in the load balance mode. It is useful if used with APIs like DBI/DBD:Pg which adds white spaces against the user's will.

You need to reload pgpool.conf if you change this value.

Backends

backend_hostname

Specifies where to connect with the PostgreSQL backend. It is used by pgpool-II to communicate with the server.

For TCP/IP communication, this parameter can take a hostname or an IP address. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP; the value is the name of the directory in which the socket file is stored. The default behavior when backend_hostname is empty (`''`) is to connect to a Unix-domain socket in `/tmp`.

Multiple backends can be specified by adding a number at the end of the parameter name (e.g.`backend_hostname0`). This number is referred to as "DB node ID", and it starts from 0. The backend which was given the DB node ID of 0 will be called "Master DB". When multiple backends are defined, the service can be continued even if the Master DB is down (not true in some modes). In this case, the youngest DB node ID alive will be the new Master DB.

Please note that the DB node which has id 0 has no special meaning if operated in streaming replication mode. Rather, you should care about if the DB node is the "primary node" or not. See Streaming Replication for more details.

If you plan to use only one PostgreSQL server, specify it by `backend_hostname0`.

New nodes can be added in this parameter by reloading a configuration file. However, values cannot be updated so you must restart pgpool-II in that case.

backend_port

Specifies the port number of the backends. Multiple backends can be specified by adding a number at the end of the parameter name (e.g. `backend_port0`). If you plan to use only one PostgreSQL server, specify it by `backend_port0`.

New backend ports can be added in this parameter by reloading a configuration file. However, values cannot be updated so you must restart pgpool-II in that case.

backend_weight

Specifies the load balance ratio for the backends. Multiple backends can be specified by adding a number at the end of the parameter name (e.g. `backend_weight0`). If you plan to use only one PostgreSQL server, specify it by `backend_weight0`. In the raw mode, set to 1.

New backend weights can be added in this parameter by reloading a configuration file.

From pgpool-II 2.2.6/2.3 or later, you can change this value by re-loading the configuration file. This will take effect only for new established client sessions. This is useful if you want to prevent any query sent to slaves to perform some administrative work in master/slave mode.

backend_data_directory

Specifies the database cluster directory of the backends. Multiple backends can be specified by adding a number at the end of the parameter name (e.g. `backend_data_directory0`). If you don't plan to use online recovery, you do not need to specify this parameter.

New backend data directories can be added in this parameter by reloading a configuration file. However, values cannot be updated so you must restart pgpool-II in that case.

backend_flag V3.1 -

Controls various backend behavior. Multiple backends can be specified by adding a number at the end of the parameter name (e.g. `backend_flag0`).

Currently followings are allowed. Multiple flags can be specified by using "|".

ALLOW_TO_FAILOVER	Allow to failover or detaching backend. This is the default. You cannot specify with DISALLOW_TO_FAILOVER at a same time.
DISALLOW_TO_FAILOVER	Disallow to failover or detaching backend. This is useful when you protect backend by using HA(High Availability) softwares such as Heartbeat or Pacemaker. You cannot specify with ALLOW_TO_FAILOVER at a same time.

SSL

ssl V2.3 -

If true, enable SSL support for both the frontend and backend connections. Note that `ssl_key` and `ssl_cert` must also be set in order for SSL to work with frontend connections.

SSL is off by default. Note that OpenSSL support must also have been configured at compilation time, as mentioned in the installation section.

The pgpool-II daemon must be restarted when updating SSL related settings.

ssl_key V2.3 -

The path to the private key file to use for incoming frontend connections.

There is no default value for this option, and if left unset SSL will be disabled for incoming frontend connections.

ssl_cert V2.3 -

The path to the public x509 certificate file to use for incoming frontend connections.

There is no default value for this option, and if left unset SSL will be disabled for incoming frontend connections.

ssl_ca_cert

The path to a PEM format file containing one or more CA root certificates, which can be used to verify the backend server certificate. This is analogous to the `-CAfile` option of the OpenSSL `verify(1)` command.

The default value for this option is unset, so no verification takes place. Verification will still occur if this option is not set but a value has been given for `ssl_ca_cert_dir`.

ssl_ca_cert_dir

The path to a directory containing PEM format CA certificate files, which can be used to verify the backend server certificate. This is analogous to the `-CApath` option of the OpenSSL `verify(1)` command.

The default value for this option is unset, so no verification takes place. Verification will still occur if this option is not set but a value has been given for `ssl_ca_cert`.

Other

relcache_expire V3.1 -

Life time of relation cache in seconds. 0 means no cache expiration(the default). The relation cache is used for cache the query result against PostgreSQL system catalog to obtain various information including table structures or if it's a temporary table or not. The cache is maintained in a pgpool child local memory and being kept as long as it survives. If someone modify the table by using ALTER TABLE or some such, the relcache is not consistent anymore. For this purpose, relcache_expiration controls the life time of the cache.

relcache_size V3.2 -

Number of relcache entries. Default is 256. If you see following message frequently, increase the number.

```
"pool_search_relcache: cache replacement happened"
```

check_temp_table V3.2 -

If on, enable temporary table check in SELECT statements. This initiates queries against system catalog of primary/master thus increases load of primary/master. If you are absolutely sure that your system never uses temporary tables and you want to save access to primary/master, you could turn this off. Default is on.

Generating SSL certificates

Certificate handling is outside the scope of this document. The Secure TCP/IP Connections with SSL page at postgresql.org has pointers with sample commands for how to generate self-signed certificates.

Failover in the raw Mode

Failover can be performed in raw mode if multiple servers are defined. pgpool-II usually accesses the backend specified by `backend_hostname0` during normal operation. If the backend_hostname0 fails for some reason, pgpool-II tries to access the backend specified by backend_hostname1. If that fails, pgpool-II tries the backend_hostname2, 3 and so on.

back to top

Connection Pool Mode

In connection pool mode, all functions in raw mode and the connection pool function can be used. To enable this mode, you need to turn on "connection_cache". Following parameters take effect to connection pool.

max_pool

The maximum number of cached connections in pgpool-II children processes. pgpool-II reuses the cached connection if an incoming connection is connecting to the same database with the same user name. If not, pgpool-II creates a new connection to the backend. If the number of cached connections exceeds max_pool, the oldest connection will be discarded, and uses that slot for the new connection.

Default value is 4. Please be aware that the number of connections from pgpool-II processes to the backends may reach num_init_children * max_pool.

This parameter can only be set at server start.

connection_life_time

Cached connections expiration time in seconds. An expired cached connection will be disconnected. Default is 0, which means the cached connections will not be disconnected.

reset_query_list

Specifies the SQL commands sent to reset the connection to the backend when exiting a session. Multiple commands can be specified by delimiting each by ";". Default is the following, but can be changed to suit your system.

```
reset_query_list = 'ABORT; DISCARD ALL'
```

Commands differ in each PostgreSQL versions. Here are the recommended settings.

PostgreSQL version	reset_query_list value
7.1 or before	ABORT

7.2 to 8.2	ABORT; RESET ALL; SET SESSION AUTHORIZATION DEFAULT
8.3 or later	ABORT; DISCARD ALL

- "ABORT" is not issued when not in a transaction block for 7.4 or later.

You need to reload pgpool.conf upon modification of this directive.

Failover in the Connection Pool Mode

Failover in the connection pool mode is the same as in the raw mode.

back to top

Replication Mode

This mode enables data replication between the backends. The configuration parameters below must be set in addition to everything above.

replication_mode

Setting to true enables replication mode. Default is false.

load_balance_mode

When set to true, SELECT queries will be distributed to each backend for load balancing. Default is false.

This parameter can only be set at server start.

replication_stop_on_mismatch

When set to true, if all backends don't return the same packet kind, the backends that differ from most frequent result set are degenerated.

A typical use case is a SELECT statement being part of a transaction, replicate_select set to true, and SELECT returning a different number of rows among backends. Non-SELECT statements might trigger this though. For example, a backend succeeded in an UPDATE, while others failed. Note that pgpool does NOT examine the content of records returned by SELECT.

If set to false, the session is terminated and the backends are not degenerated. Default is false.

failover_if_affected_tuples_mismatch V3.0 -

When set to true, if backends don't return the same number of affected tuples during an INSERT/UPDATE /DELETE, the backends that differ from most frequent result set are degenerated. If the frequencies are

same, the group which includes master DB node (a DB node having the youngest node id) is remained and other groups are degenerated.

If set to false, the session is terminated and the backends are not degenerated. Default is false.

white_function_list V3.0 -

Specify a comma separated list of function names that **do not** update the database. SELECTs using functions not specified in this list are neither load balanced, nor replicated if in replication mode. In master slave mode, such SELECTs are sent to master (primary) only.

You can use regular expression into the list to match function name (to which added automatically ^ and $), for example if you have prefixed all your read only function with 'get_' or 'select_'

```
white_function_list = 'get_.*,select_.*'
```

black_function_list V3.0 -

Specify a comma separated list of function names that **do** update the database. SELECTs using functions specified in this list are neither load balanced, nor replicated if in replication mode. In master slave mode, such SELECTs are sent to master(primary) only.

You can use regular expression into the list to match function name (to which added automatically ^ and $) for example if you have prefixed all your updating functions with 'set_', 'update_', 'delete_' or 'insert_':

```
black_function_list = 'nextval,setval,set_.*,update_.*,delete_.*,insert_.*'
```

Only one of these two lists can be filled in a configuration.

Prior to pgpool-II 3.0, nextval() and setval() were known to do writes to the database. You can emulate this by using white_function_list and black_function_list:

```
white_function_list = ''
black_function_list = 'nextval,setval,lastval,currval'
```

Please note that we have lastval and currval in addition to nextval and setval. Though lastval() and currval() are not writing functions, it is wise to add lastval() and currval() to avoid errors in the case when these functions are accidentally load balanced to other DB node. Because adding to black_function_list will prevent load balancing.

replicate_select

When set to true, pgpool-II replicates SELECTs in replication mode. If false, pgpool-II only sends them to the Master DB. Default is false.

If a SELECT query is inside an explicit transaction block, replicate_select and load_balance_mode will have an effect on how replication works. Details are shown below.

SELECT is inside a transaction block	Y	Y	Y	N	N	N	Y	N
replicate_select is true	Y	Y	N	N	Y	Y	N	N
load_balance_mode is true	Y	N	N	N	Y	N	Y	Y
results(R:replication, M: send only to master, L: load balance)	R	R	M	M	R	R	M	L

insert_lock

If replicating a table with SERIAL data type, the SERIAL column value may differ between the backends. This problem is avoidable by locking the table explicitly (although, transactions' parallelism will be severely degraded). To achieve this, however, the following change must be made:

```
INSERT INTO ...
```

to

```
BEGIN;
LOCK TABLE ...
INSERT INTO ...
COMMIT;
```

When `insert_lock` is true, pgpool-II automatically adds the above queries each time an INSERT is executed (if already in transaction, it simply adds LOCK TABLE ....).

pgpool-II 2.2 or later, it automatically detects whether the table has a SERIAL columns or not, so it will never lock the table if it does not use SERIAL columns.

pgpool-II 3.0 series until 3.0.4 uses a row lock against the sequence relation, rather than table lock. This is intended to minimize lock conflict with VACUUM (including autovacuum). However this will lead to another problem. After transaction wraparound happens, row locking against the sequence relation causes PostgreSQL internal error (more precisely, access error on pg_clog, which keeps transaction status). To prevent this, PostgreSQL core developers decided to disallow row locking against sequences and this will break pgpool-II of course (the "fixed" version of PostgreSQL was released as 9.0.5, 8.4.9, 8.3.16 and 8.2.22).

pgpool-II 3.0.5 or later uses a row lock against pgpool_catalog.insert_lock table because new PostgreSQL disallows a row lock against the sequence relation. So creating insert_lock table in all databases which are accessed via pgpool-II beforehand is required. See Creating insert_lock table for more details. If does not exist insert_lock table, pgpool-II locks the insert target table. This behavior is same as pgpool-II 2.2 and 2.3 series. If you want to use insert_lock which is compatible with older releases, you can specify lock method by configure script. See configure for more details.

You might want to have a finer (per statement) control:

1. set `insert_lock` to true, and add `/*NO INSERT LOCK*/` at the beginning of an INSERT statement for which you do not want to acquire the table lock.
2. set `insert_lock` to false, and add `/*INSERT LOCK*/` at the beginning of an INSERT statement for which you want to acquire the table lock.

Default value is false. If `insert_lock` is enabled, the regression tests for PostgreSQL 8.0 will fail in transactions, privileges, rules, and alter_table. The reason for this is that pgpool-II tries to LOCK the VIEW for the rule test, and will produce the following error message:

```
! ERROR: current transaction is aborted, commands ignored until
end of transaction block
```

For example, the transactions test tries an INSERT into a table which does not exist, and pgpool-II causes PostgreSQL to acquire the lock before that. The transaction will be aborted, and the following INSERT statement produces the above error message.

recovery_user

This parameter specifies a PostgreSQL user name for online recovery. It can be changed without restarting.

recovery_password

This parameter specifies a PostgreSQL password for online recovery. It can be changed without restarting.

recovery_1st_stage_command

This parameter specifies a command to be run by master(primary) PostgreSQL server at the first stage of online recovery. The command file must be put in the database cluster directory for security reasons. For example, if recovery_1st_stage_command = 'sync-command', then pgpool-II executes $PGDATA/sync-command.

recovery_1st_stage_command will receive 3 parameters as follows:

1. path to master(primary) database cluster
2. PostgreSQL host name to be recovered
3. path to database cluster to be recovered

Note that pgpool-II **accepts** connections and queries while recovery_1st_stage command is executed. You can retrieve and update data during this stage.

This parameter can be changed without restarting.

recovery_2nd_stage_command

This parameter specifies a command to be run by master(primary) PostgreSQL server at the second stage of online recovery. The command file must be put in the database cluster directory for security reasons. For example, if recovery_2nd_stage_command = 'sync-command', then pgpool-II executes $PGDATA/sync-command.

recovery_2nd_stage_command will receive 3 parameters as follows:

1. path to master(primary) database cluster
2. PostgreSQL host name to be recovered

3. path to database cluster to be recovered

Note that pgpool-II **does not accept** connections and queries while recovery_2nd_stage_command is running. Thus if a client stays connected for a long time, the recovery command won't be executed. pgpool-II waits until all clients have closed their connections. The command is only executed when no client is connected to pgpool-II anymore.

This parameter can be changed without restarting.

recovery_timeout

pgpool does not accept new connections during the second stage. If a client connects to pgpool during recovery processing, it will have to wait for the end of the recovery.

This parameter specifies recovery timeout in sec. If this timeout is reached, pgpool cancels online recovery and accepts connections. 0 means no wait.

This parameter can be changed without restarting.

client_idle_limit_in_recovery V2.2 -

Similar to client_idle_limit but only takes effect in the second stage of recovery. A client being idle for client_idle_limit_in_recovery seconds since its last query will get disconnected. This is useful for preventing the pgpool recovery from being disturbed by a lazy client or if the TCP/IP connection between the client and pgpool is accidentally down (a cut cable for instance). If set to -1, disconnect the client immediately. The default value for client_idle_limit_in_recovery is 0, which means the feature is turned off.

If your clients are very busy, pgpool-II cannot enter the second stage of recovery whatever value of client_idle_limit_in_recovery you may choose. In this case, you can set client_idle_limit_in_recovery to -1 so that pgpool-II immediately disconnects such busy clients before entering the second stage.

You need to reload pgpool.conf if you change client_idle_limit_in_recovery.

lobj_lock_table V2.2 -

This parameter specifies a table name used for large object replication control. If it is specified, pgpool will lock the table specified by lobj_lock_table and generate a large object id by looking into pg_largeobject system catalog and then call lo_create to create the large object. This procedure guarantees that pgpool will get the same large object id in all DB nodes in replication mode. Please note that PostgreSQL 8.0 or older does not have lo_create, thus this feature will not work.

A call to the libpq function lo_creat() will trigger this feature. Also large object creation through Java API (JDBC driver), PHP API (pg_lo_create, or similar API in PHP library such as PDO), and this same API in various programming languages are known to use a similar protocol, and thus should work.

The following large object create operation will not work:

- lo_create of libpq
- Any API of any language using lo_create
- lo_import function in backend
- SELECT lo_create

It does not matter what schema lobj_lock_table is stored in, but this table should be writable by any user. Here is an example showing how to create such a table:

```
CREATE TABLE public.my_lock_table ();
GRANT ALL ON public.my_lock_table TO PUBLIC;
```

The table specified by lobj_lock_table must be created beforehand. If you create the table in template1, any database created afterward will have it.

If lobj_lock_table has empty string(''), the feature is disabled (thus large object replication will not work). The default value for lobj_lock_table is ''.

condition for load balancing

For a query to be load balanced, all the following requirements must be met:

- PostgreSQL version 7.4 or later
- either in replication mode or master slave mode
- the query must not be in an explicitly declared transaction (i.e. not in a BEGIN ~ END block) when operated in replication mode
- it's not SELECT INTO
- it's not SELECT FOR UPDATE nor FOR SHARE
- it starts with "SELECT" or one of COPY TO STDOUT, EXPLAIN, EXPLAIN ANALYZE SELECT... ignore_leading_white_space = true will ignore leading white space. (Except for SELECTs using writing functions specified in black_list or white_list)
- V3.0 - in master slave mode, in addition to above, following conditions must be met:
  - does not use temporary tables
  - does not use unlogged tables
  - does not use system catalogs
  - However, if following conditions are met, load balance is possible even if in an explicit transaction
    - transaction isolation level is not SERIALIZABLE
    - the transaction has not issued a write query yet(until a write query issued, load balance is possible)

Note that you could suppress load balancing by inserting arbitrary comments just in front of the SELECT query:

```
/*REPLICATION*/ SELECT ...
```

Please refer to replicate_select as well. See also a flow chart.

Note: the JDBC driver has an autocommit option. If autocommit is false, the JDBC driver sends "BEGIN" and "COMMIT" by itself. So pgpool cannot do any load balancing. You need to call setAutoCommit(true) to enable autocommit.

Failover in Replication Mode

pgpool-II degenerates a dead backend and continues the service. The service can be continued if there is at least one backend alive.

Specific errors in replication mode

In replication mode, if pgpool finds that the number of affected tuples by INSERT, UPDATE, DELETE are not same, it sends erroneous SQL statement to all DB nodes to abort the transaction if failover_if_affected_tuples_mismatch is set to false (degeneration occurs if it is set to true). In this case you will see following error messages on client terminal:

```
=# UPDATE t SET a = a + 1;
ERROR: pgpool detected difference of the number of update tuples Possible last query was: "update t1 set i = 1;"
HINT: check data consistency between master and other db node
```

You will see number of updated rows in PostgreSQL log (in this case DB node 0 has 0 updated row and DB node 1 has 1 updated row)

```
2010-07-22 13:23:25 LOG:   pid 5490: SimpleForwardToFrontend: Number of affected tuples are: 0 1
2010-07-22 13:23:25 LOG:   pid 5490: ReadyForQuery: Degenerate backends: 1
2010-07-22 13:23:25 LOG:   pid 5490: ReadyForQuery: Number of affected tuples are: 0 1
```

back to top

Master/Slave Mode

This mode is used to couple pgpool-II with another master/slave replication software (like Slony-I and Streaming replication), which is responsible for doing the actual data replication. DB nodes' information (backend_hostname, backend_port, backend_weight, backend_flag and backend_data_directory if you need the online recovery functionality) must be set, in the same way as in the replication mode. In addition to that, set master_slave_mode and load_balance_mode to true.

pgpool-II will then send queries that need to be replicated to the Master DB, and other queries will be load balanced if possible. Queries sent to Master DB because they cannot be balanced are of course accounted for in the load balancing

algorithm.

In master/slave mode, DDL and DML for temporary table can be executed on the master node only. SELECT can be forced to be executed on the master as well, but for this you need to put a /*NO LOAD BALANCE*/ comment before the SELECT statement.

In the master/slave mode, replication_mode must be set to false, and master_slave_mode to true.

The master/slave mode has a 'master_slave_sub mode'. The default is 'slony' which is suitable for Slony-I. You can also set it to 'stream', which should be set if you want to work with PostgreSQL's built-in replication system (Streaming Replication). The sample configuration file for the Slony-I sub-mode is pgpool.conf.sample-master-slave and the sample for the streaming replication sub-module is pgpool.conf.sample-stream.

Please restart pgpool-II if you change any of the above parameters.

You can set white_function_list and black_function_list to control load balancing in master/slave mode. See white_function_list for more details.

back to top

Streaming Replication V3.1 -

As stated above, pgpool-II can work together with Streaming Replication, which is available since PostgreSQL 9.0. To use it, enable 'master_slave_mode' and set 'master_slave_sub_mode' to 'stream'. pgpool-II assumes that Streaming Replication is used with Hot Standby at present, which means that the standby database is open read-only. The following directives can be used with this mode:

delay_threshold V3.0 -

Specifies the maximum tolerated replication delay of the standby against the primary server in WAL bytes. If the delay exceeds delay_threshold, pgpool-II does not send SELECT queries to the standby server anymore. Everything is sent to the primary server even if load balance mode is enabled, until the standby has caught-up. If delay_threshold is 0 or sr checking is disabled, the delay checking is not performed. This check is performed every 'sr_check_period'. The default value for delay_threshold is 0.

You need to reload pgpool.conf if you change this directive.

sr_check_period V3.1 -

This parameter specifies the interval between the streaming replication delay checks in seconds. Default is 0, which means the check is disabled.

You need to reload pgpool.conf if you change sr_check_period.

sr_check_user V3.1 -

The user name to perform streaming replication check. This user must exist in all the PostgreSQL backends. Otherwise, the check causes an error. Note that sr_check_user and sr_check_password are used even

sr_check_period is 0. To identify the primary server, pgpool-II sends function call request to each backend. sr_check_user and sr_check_password are used for this session.

You need to reload pgpool.conf if you change sr_check_user.

sr_check_password V3.1 -

The password of the user to perform streaming replication check. If no password is required, specify empty string('').

You need to reload pgpool.conf if you change sr_check_password.

log_standby_delay V3.1 -

Specifies how to log the replication delay. If 'none' is specified, no log is written. If 'always', log the delay every time health checking is performed. If 'if_over_threshold' is specified, the log is written when the delay exceeds delay_threshold. The default value for log_standby_delay is 'none'. You need to reload pgpool.conf if you change this directive.

You could monitor the replication delay by using the "show pool_status" command as well. The column name is "standby_delay#"(where '#' should be replaced by DB node id).

Failover with Streaming Replication

In master/slave mode with streaming replication, if the primary or standby node goes down, pgpool-II can be set up to trigger a failover. Nodes can be detached automatically without further setup. While doing streaming replication, the standby node checks for the presence of a "trigger file" and on finding it, the standby stops continuous recovery and goes into read-write mode. By using this, you can have the standby database take over when the primary goes down.

Caution: If you plan to use multiple standby nodes, we recommend to set a delay_threshold to prevent any query directed to other standby nodes from retrieving older data.

If a second standby took over primary when the first standby has already taken over too, you would get bogus data from the second standby. We recommend not to plan this kind of configuration.

How to setup a failover configuration is as follows.

1. Put a failover script somewhere (for example /usr/local/pgsql/bin) and give it execute permission.

```
$ cd /usr/loca/pgsql/bin
$ cat failover_stream.sh
#! /bin/sh
# Failover command for streaming replication.
# This script assumes that DB node 0 is primary, and 1 is standby.
#
# If standby goes down, do nothing. If primary goes down, create a
# trigger file so that standby takes over primary node.
#
# Arguments: $1: failed node id. $2: new master hostname. $3: path to
```

```
# trigger file.

failed_node=$1
new_master=$2
trigger_file=$3

# Do nothing if standby goes down.
if [ $failed_node = 1 ]; then
    exit 0;
fi

# Create the trigger file.
/usr/bin/ssh -T $new_master /bin/touch $trigger_file

exit 0;

chmod 755 failover_stream.sh
```

2. Set failover_commmand in pgpool.conf.

```
failover_command = '/usr/local/src/pgsql/9.0-beta/bin/failover_stream.sh %d %H /tmp/trigger_file0'
```

3. Set recovery.conf on the standby node. A sample recovery.conf can be found under the PostgreSQL installation directory. Its name is "share/recovery.conf.sample". Copy recovery.conf.sample as recovery.conf inside the database cluster directory and edit it.

```
standby_mode = 'on'
primary_conninfo = 'host=name of primary_host user=postgres'
trigger_file = '/tmp/trigger_file0'
```

4. Set postgresql.conf on the primary node. Below is just an example. You will need to tweak it for your environment.

```
wal_level = hot_standby
max_wal_senders = 1
```

5. Set pg_hba.conf on the primary node. Below is just an example. You will need to tweak it for your environment.

```
host    replication    postgres        192.168.0.10/32        trust
```

Start primary and secondary PostgreSQL nodes to initiate Streaming replication. If the primary node goes down, the standby node will automatically start as a normal PostgreSQL and will be ready to accept write queries.

Streaming Replication

While using Streaming replication and Hot Standby, it is important to determine which query can be sent to the primary or the standby, and which one should not be sent to the standby. pgpool-II's Streaming Replication mode carefully takes care of this. In this chapter we'll explain how pgpool-II accomplishes this.

We distinguish which query should be sent to which node by looking at the query itself.

- These queries should be sent to the primary node only
  - INSERT, UPDATE, DELETE, COPY FROM, TRUNCATE, CREATE, DROP, ALTER, COMMENT
  - SELECT ... FOR SHARE | UPDATE
  - SELECT in transaction isolation level SERIALIZABLE
  - LOCK command more strict than ROW EXCLUSIVE MODE
  - Some transactional commands:
    - BEGIN READ WRITE, START TRANSACTION READ WRITE
    - SET TRANSACTION READ WRITE, SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
    - SET transaction_read_only = off
  - Two phase commit commands: PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED
  - LISTEN, UNLISTEN, NOTIFY
  - VACUUM
  - Some sequence functions (nextval and setval)
  - Large objects creation commands
- These queries can be sent to both the primary node and the standby node. If load balancing is enabled, these types of queries can be sent to the standby node. However, if delay_threshold is set and the replication delay is higher than delay_threshold, queries are sent to the primary node.
  - SELECT not listed above
  - COPY TO
  - DECLARE, FETCH, CLOSE
  - SHOW
- These queries are sent to both the primary node and the standby node
  - SET
  - DISCARD
  - DEALLOCATE ALL

In an explicit transaction:

- Transaction starting commands such as BEGIN are sent to the primary node.
- Following SELECT and some other queries that can be sent to both primary or standby are executed in the transaction or on the standby node.
- Commands which cannot be executed on the standby such as INSERT are sent to the primary. After one of these commands, even SELECTs are sent to the primary node, This is because these SELECTs might want to see the result of an INSERT immediately. This behavior continues until the transaction closes or aborts.

In the extended protocol, it is possible to determine if the query can be sent to standby or not in load balance mode while parsing the query. The rules are the same as for the non extended protocol. For example, INSERTs are sent to the primary node. Following bind, describe and execute will be sent to the primary node as well.

[Note: If the parse of a SELECT statement is sent to the standby node due to load balancing, and then a DML statement, such as an INSERT, is sent to pgpool-II, then the parsed SELECT will have to be executed on the primary node. Therefore, we re-parse the SELECT on the primary node.]

Lastly, queries that pgpool-II's parser thinks to be an error are sent to the primary node.

Online recovery with Streaming Replication

In master/slave mode with streaming replication, online recovery can be performed. In the online recovery procedure, primary server acts as a master server and recovers specified standby server. Thus the recovery procedure requires that the primary server is up and running. If the primary server goes down, and no standby server is promoted, you need to stop pgpool-II and all PostgreSQL servers and recover them manually.

1. Set recovery_user. Usually it's "postgres".

```
recovery_user = 'postgres'
```

2. Set recovery_password for recovery_user to login database.

```
recovery_password = 't-ishii'
```

3. Set recovery_1st_stage_command. The script for this stage should perform a base backup of the primary and restore it on the standby node. Place this script inside the primary database cluster directory and give it execute permission. Here is the sample script (basebackup.sh) for a configuration of one primary and one standby. You need to setup ssh so that recovery_user can login from the primary to the standby without being asked for a password.

```
recovery_1st_stage_command = 'basebackup.sh'
```

4. Leave recovery_2nd_stage_command be empty.

```
recovery_2nd_stage_command = ''
```

5. Install required C and SQL functions to perform online recovery into each DB nodes.

```
# cd pgpool-II-x.x.x/sql/pgpool-recovery
# make
# make install
# psql -f pgpool-recovery.sql template1
```

6. After completing online recovery, pgpool-II will start PostgreSQL on the standby node. Install the script for this purpose on each DB nodes. Sample script is included in "sample" directory of the source code. This script uses ssh. You need to allow recovery_user to login from the primary node to the standby node without being asked password.

That's it. Now you should be able to use pcp_recovery_node (as long as the standby node stops) or push "recovery" button of pgpoolAdmin to perform online recovery. If something goes wrong, please examine pgpool-II log, primary server log and standby server log(s).

For your reference, here are the steps taken in the recovery procedure.

1. Pgpool-II connects to primary server's template1 database as user = recovery_user, password = recovery_password.
2. Primary server executes pgpool_recovery function.
3. pgpool_recovery function executes recovery_1st_stage_command. Note that PostgreSQL executes functions with database cluster as the current directory. Thus recovery_1st_stage_command is executed in the database cluster directory.
4. Primary server executes pgpool_remote_start function. This function executes a script named "pgpool_remote_start" in the database cluster directory, and it executes pg_ctl command on the standby server to be recovered via ssh. pg_ctl will start postmaster in background. So we need to make sure that postmaster on the standby actually starts.
5. pgpool-II tries to connect to the standby PostgreSQL as user = recovery_user and password = recovery_password. The database to be connected is "postgres" if possible. Otherwise "template1" is used. pgpool-II retries for recovery_timeout seconds. If success, go to next step.
6. If failback_command is not empty, pgpool-II parent process executes the script.
7. After failback_command finishes, pgpool-II restart all child processes.

back to top

Parallel Mode

This mode activates parallel execution of queries. Tables can be split, and data distributed to each node. Moreover, the replication and the load balancing features can be used at the same time. In parallel mode, replication_mode and load_balance_mode are set to true in pgpool.conf, master_slave is set to false, and parallel_mode is set to true. When you change this parameter, restart pgpool-II.

Configuring the System DB

To use the parallel mode, the System DB must be configured properly. The System DB contains rules, stored in a table, to choose an appropriate backend to send partitioned data to. The System DB does not need to be created on the same host as pgpool-II. The System DB's configuration is done in `pgpool.conf`.

system_db_hostname

The hostname where the System DB is. Specifying the empty string ('') means the System DB is on the same host as pgpool-II, and will be accessed via a UNIX domain socket.

system_db_port

The port number for the System DB

system_dbname

The partitioning rules and other information will be defined in the database specified here. Default value is: `'pgpool'`.

system_db_schema

The partitioning rules and other information will be defined in the schema specified here. Default value is: `'pgpool_catalog'`.

system_db_user

The user name to connect to the System DB.

system_db_password

The password for the System DB. If no password is necessary, set the empty string ('').

Initial Configuration of the System DB

First, create the database and schema specified in the `pgpool.conf` file. A sample script can be found in `$prefix/share/system_db.sql`. If you have specified a different database name or schema, change them accordingly in the script.

```
psql -f $prefix/share/system_db.sql pgpool
```

Registering a Partitioning Rule

The rules for data partitioning must be registered into the `pgpool_catalog.dist_def` table.

```
CREATE TABLE pgpool_catalog.dist_def(
    dbname TEXT,                                          -- database name
    schema_name TEXT,                                     -- schema name
    table_name TEXT,                                      -- table name
    col_name TEXT NOT NULL CHECK (col_name = ANY (col_list)), -- partitioning key column name
    col_list TEXT[] NOT NULL,                             -- names of table attributes
    type_list TEXT[] NOT NULL,                            -- types of table attributes
    dist_def_func TEXT NOT NULL,                          -- name of the partitioning rule function
    PRIMARY KEY (dbname,schema_name,table_name)
);
```

Registering a Replication Rule

Tables that are not distributed have to be replicated. When a query joins a distributed table with another table, pgpool gets the replication information from the pgpool_catalog.replicate_def table. A table has to be either replicated or distributed.

```
CREATE TABLE pgpool_catalog.replicate_def(
    dbname TEXT,                  -- database name
    schema_name TEXT,             -- schema name
    table_name TEXT,              --table name
    col_list TEXT[] NOT NULL,     -- names of table attributes
    type_list TEXT[] NOT NULL,    -- types of table attributes
    PRIMARY KEY (dbname,schema_name,table_name)
);
```

Example for partitioning the pgbench tables

In this example, the accounts table is partitioned, and the branches and tellers table are replicated. The accounts table and the branches table are joined by bid. The branches table is registered into the replication table. If the three tables (accounts, branches, and tellers) are to be joined, it is necessary to register a replication rule for the tellers table too.

```
INSERT INTO pgpool_catalog.dist_def VALUES (
    'pgpool',
    'public',
    'accounts',
    'aid',
    ARRAY['aid','bid','abalance','filler'],
    ARRAY['integer','integer','integer','character(84)'],
    'pgpool_catalog.dist_def_accounts'
);

INSERT INTO pgpool_catalog.replicate_def VALUES (
    'pgpool',
    'public',
    'branches',
    ARRAY['bid','bbalance','filler'],
    ARRAY['integer','integer','character(84)']
);
```

The partitioning rule function (here, pgpool_catalog.dist_def_accounts) takes a value for the partitioning key column, and returns the corresponding DB node ID. Note that the node ID must start from 0. Below is an example of this function for pgbench.

```
CREATE OR REPLACE FUNCTION pgpool_catalog.dist_def_accounts (val ANYELEMENT) RETURNS INTEGER AS '
SELECT CASE WHEN $1 >= 1 and $1 <= 30000 THEN 0
WHEN $1 > 30000 and $1 <= 60000 THEN 1
ELSE 2
```

back to top

Setting up pool_hba.conf for client authentication (HBA)

Just like the pg_hba.conf file for PostgreSQL, pgpool supports a similar client authentication function using a configuration file called "pool_hba.conf".

When you install pgpool, pool_hba.conf.sample will be installed in "/usr/local/etc", which is the default directory for configuration files. Copy pool_hba.conf.sample as pool_hba.conf and edit it if necessary. By default, pool_hba authentication is disabled. Change enable_pool_hba to on to enable it.

The format of the pool_hba.conf file follows very closely PostgreSQL's pg_hba.conf format.

```
local      DATABASE  USER  METHOD  [OPTION]
host       DATABASE  USER  CIDR-ADDRESS  METHOD  [OPTION]
```

See "pool_hba.conf.sample" for a detailed explanation of each field.

Here are the limitations of pool_hba.

- "hostssl" connection type is not supported

  Even if the SSL support is enabled, "hostssl" cannot be used. See SSL for more details.

- "samegroup" for DATABASE field is not supported

  Since pgpool does not know anything about users in the backend server, the database name is simply checked against entries in the DATABASE field of pool_hba.conf.

- group names following "+" for USER field is not supported

  This is for the same reason as for the "samegroup" described above. A user name is simply checked against the entries in the USER field of pool_hba.conf.

- IPv6 for IP address/mask is not supported

  pgpool currently does not support IPv6.

- Only "trust", "reject", "md5" and "pam" for METHOD field are supported

  Again, this is for the same reason as for the "samegroup" described above. pgpool does not have access to user/password information.

  To use md5 authentication, you need to register your name and password in "pool_passwd". See Authentication / Access Controls for more details.

Note that everything described in this section is about the authentication taking place between a client and pgpool; a client still has to go through the PostgreSQL's authentication process. As far as pool_hba is concerned, it does not matter if a user name and/or database name given by a client (i.e. psql -U testuser testdb) really exists in the backend. pool_hba only cares if a match in the pool_hba.conf is found or not.

PAM authentication is supported using user information on the host where pgpool is executed. To enable PAM support in pgpool, specify "--with-pam" option to configure:

```
configure --with-pam
```

To enable PAM authentication, you need to create a service-configuration file for pgpool in the system's PAM configuration directory (which is usually at "/etc/pam.d"). A sample service-configuration file is installed as "share/pgpool.pam" under the install directory.

back to top

Setting Query cache method - V3.1 (DEPRECATED)

Caution: this (on disk) query cache functionality will be removed in the near future. Please use on memory query cache instead.

The Query cache can be used in all modes in pgpool-II. The query cache allow to reuse the SELECT result to boost the performance. Activating it in pgpool.conf is done as follows:

```
enable_query_cache = true
```

You'll have to create the following table in the System DB too:

```
CREATE TABLE pgpool_catalog.query_cache (
  hash TEXT,
  query TEXT,
  value bytea,
  dbname TEXT,
  create_time TIMESTAMP WITH TIME ZONE,
  PRIMARY KEY(hash, dbname)
);
```

However, you may have to modify the schema in this statement, if you don't use "pgpool_catalog".

Caution: Current query cache implementation creates cache data on database. Thus enabling query cache may not contribute to boost performance. Contents of query cache is not updated even if the underlying table is get updated. You need to delete the cache data from the cache table or restart pgpool-II with -c (delete cache) option.

back to top

On memory query Cache V3.2 -

You can use on memory query cache in any mode. It is different from the above query cache on the point that on memory query cache is faster because cache storage is on memory. Moreover you don't need to restart pgpool-II when the cache is outdated because the underlying table gets updated.

On memory cache saves pair of SELECT statements (with its Bind parameters if the SELECT is an extended query). If

the same SELECTs comes in, it returns the value from cache. Since no SQL parsing nor access to PostgreSQL are involved, it's extremely fast.

On the other hand, it might be slower than the normal path because it adds some overhead to store cache. Moreover when a table is updated, pgpool automatically deletes all the caches related to the table. So the performance will be degraded by a system with a lot of updates. If the cache_hit_ratio is lower than 70%, you might want to disable on memory cache.

Restrictions

- On memory query cache deletes the all cache of an updated table automatically with monitoring if the executed query is UPDATE, INSERT, ALTER TABLE and so on. But pgpool-II isn't able to recognize implicit updates due to triggers, foreign keys and DROP TABLE CASCADE. You can avoid this problem with memqcache_expire by which pgpool deletes old cache in a fixed time automatically, or with black_memqcache_table_list by which pgpool's memory cache flow ignores the tables.
- If you want to use multiple instances of pgpool-II with online memory cache which uses shared memory, it could happen that one pgpool deletes cache, and the other one doesn't do it thus finds old cached result when a table gets updated. Memcached is the better cache storage in this case.

Enabling on memory query cache

To enable the memory cache functionality, set this to on (default is off).

```
memory_cache_enabled = on
```

Choosing cache storage

You can choose a cache storage: shared memory or memcached (you can't use the both). Query cache with shared memory is fast and easy because you don't have to install and configure memcached, but restricted the max size of cache by the one of shared memory. Query cache with memcached needs a overhead to access network, but you can set the size as you like.

Memory cache behavior can be specified by memqcache_method directive. Either "shmem"(shared memory) or "memcached". Default is shmem.

```
memqcache_method = 'shmem'
```

When on memory query cache is prohibited

Not All of SELECTs and WITH can be cached. In some cases including followings, cache is avoided to keep consistency between caches and databases.

- SELECT starting with "/*NO QUERY CACHE*/" comment
- SELECT including tables in black_memqcache_table_list
- SELECT FOR SHARE / UPDATE
- SELECT including un-immutable functions
- SELECT including TEMP TABLE
- SELECT including system catalogs
- SELECT including VIEWs or unlogged tables. However if the table is in white_memqcache_table_list, the result will be cached.
- SELECT including VIEWs
- SELECT in an aborted explicit transaction
- SELECT with the result larger than memqcache_maxcache

When cache is not used

It can happen that even if the matched query cache exists, pgpool doesn't return it.

- If an updating query is executed in an explicit transaction, during the transaction, pgpool doesn't use any query cache.
- The matched query cache is made by the other user (for security reason)
- The matched query cache has to be deleted due to memqcache_expire.

Configuring

These are the parameters used with both of shmem and memcached.

memqcache_expire V3.2 -

Life time of query cache in seconds. Default is 0. 0 means no cache expiration, and cache have been enabled until a table is updated. This parameter and memqcache_auto_cache_invalidation are orthogonal.

memqcache_auto_cache_invalidation V3.2 -

If on, automatically deletes cache related to the updated tables. If off, does not delete caches. Default is on. This parameter and memqcache_expire. are orthogonal.

memqcache_maxcache V3.2 -

If the size of a SELECT result is larger than memqcache_maxcache bytes, it is not cached and the messages is shown:

```
2012-05-02 15:08:17 LOG:   pid 13756: pool_add_temp_query_cache: data size exceeds memqcache_maxcache. curren
```

To avoid this problem, you have to set memqcache_maxcache larger. But if you use shared memory as the cache storage, it must be lower than memqcache_cache_block_size. If memqcached, it must be lower than the size of slab (default is 1 MB).

white_memqcache_table_list V3.2 -

Specify a comma separated list of table names whose SELECT results are to be cached even if they are VIEWs or unlogged tables. You can use regular expression (to which added automatically ^ and $).

TABLEs and VIEWs in both of white_memqcache_table_list and black_memqcache_table_list are cached.

You need to add both non schema qualified name and schema qualified name if you plan to use both of them in your query. For exmaple, if you want to use both "table1" and "public.table1" in your query, you need to add "table1,public.table1", not just "table1".

black_memqcache_table_list V3.2 -

Specify a comma separated list of table names whose SELECT results are **NOT** to be cached. You can use regular expression (to which added automatically ^ and $).

You need to add both non schema qualified name and schema qualified name if you plan to use both of them in your query. For exmaple, if you want to use both "table1" and "public.table1" in your query, you need to add "table1,public.table1", not just "table1".

memqcache_oiddir V3.2 -

Full path to the directory where oids of tables used by SELECTs are stored. Under memqcache_oiddir there are directories named database oids, and under each of them there are files named table oids used by SELECTs. In the file pointers to query cache are stored. They are used as keys to delete caches.

Directories and files under memqcache_oiddir are not deleted whenever pgpool-II restarts. If you start pgpool by "pgpool -C", pgpool starts without the old oidmap.

Monitoring caches

This explains how to monitor on memory query cache. To know if a SELECT result is from query cache or not, enable log_per_node_statement.

```
2012-05-01 15:42:09 LOG:   pid 20181: query result fetched from cache. statement: select * from t1;
```

pool_status command shows the cache hit ratio.

```
memqcache_stats_start_time          | Tue May  1 15:41:59 2012 | Start time of query cache stats
memqcache_no_cache_hits             | 80471                    | Number of SELECTs not hitting query cache
memqcache_cache_hits                | 36717                    | Number of SELECTs hitting query cache
```

In this example, you can calculate like the below:

```
(memqcache_cache_hits) / (memqcache_no_cache_hits+memqcache_cache_hits) = 36717 / (36717 + 80471) = 31.3%
```

show pool_cache commands shows the same one.

Configuring to use shared memory

These are the parameters used with shared memory as the cache storage.

memqcache_total_size V3.2 -

Specify the size of shared memory as cache storage in bytes.

memqcache_max_num_cache V3.2 -

Specify the number of cache entries. This is used to define the size of cache management space (you need this in addition to memqcache_total_size). The management space size can be calculated by: memqcache_max_num_cache * 48 bytes. Too small number will cause an error while registering cache. On the other hand too large number is just a waste of space.

memqcache_cache_block_size V3.2 -

If cache storage is shared memory, pgpool uses the memory divided by memqcache_cache_block_size. SELECT result is packed into the block. However because the SELECT result cannot be placed in several blocks, it cannot be cached if it is larger than memqcache_cache_block_size. memqcache_cache_block_size must be greater or equal to 512.

Configuring to use memcached

These are the parameters used with memcached as the cache storage.

memqcache_memcached_host V3.2 -

Specify the host name or the IP address in which memcached works. If it is the same one as pgpool-II, set 'localhost'.

memqcache_memcached_port V3.2 -

Specify the port number of memcached. Default is 11211.

memcached Installation

To use memcached as cache storage, pgpool-II needs a working memcached and the client library: libmemcached. It is easy to install them by rpms. This explains how to install from source codes.

memcached's source code can be downloaded from: memcached development page

configure

After extracting the source tarball, execute the configure script.

```
./configure
```

make

```
make
make install
```

libmemcached Installation

Libmemcached is a client library for memcached. You need to install libmemcached after installing memcached.

libmemcached's source code can be downloaded from: libmemcached development page

configure

After extracting the source tarball, execute the configure script.

```
./configure
```

If you want non-default values, some options can be set:

- `--with-memcached=path`
  The top directory where Memcached are installed.

make

```
make
```

```
make install
```

back to top

Starting/Stopping pgpool-II

Start pgpool-II

All the backends and the System DB (if necessary) must be started before starting pgpool-II.

```
pgpool [-c][-f config_file][-a hba_file][-F pcp_config_file][-n][-D][-d]
```

-c	--clear-cache	deletes query cache
-f config_file	--config-file config-file	specifies pgpool.conf
-a hba_file	--hba-file hba_file	specifies pool_hba.conf
-F pcp_config_file	--pcp-password-file	specifies pcp.conf
-n	--no-daemon	no daemon mode (terminal is not detached)
-D	--discard-status	Discard pgpool_status file and do not restore previous status V3.0 -
-C	--clear-oidmaps	Discard oid maps in memqcache_oiddir for on memory query cache (only when memqcache_method is 'memcached', if shmem, discard whenever pgpool starts). V3.2 -
-d	--debug	debug mode

Stop pgpool-II

There are two ways to stop pgpool-II. One is using a PCP command (described later), the other using a pgpool-II command. Below is an example of the pgpool-II command.

```
pgpool [-f config_file][-F pcp_config_file] [-m {s[mart]|f[ast]|i[mmediate]}] stop
```

-m s[mart]	--mode s[mart]	waits for clients to disconnect, and shutdown (default)
-m f[ast]	--mode f[ast]	does not wait for clients; shutdown immediately
-m i[mmediate]	--mode i[mmediate]	the same as '-m f'

pgpool records backend status into the [logdir]/pgpool_status file. When pgpool restarts, it reads this file and restores the backend status. This will prevent a difference in data among DB nodes which might be caused by following scenario:

1. A backend suddenly stops and pgpool executes the fail over procedure
2. An update occurs on one of the active DBs through pgpool
3. The administrator decides to stop pgpool
4. Someone decides to restart the stopping DB without notifying the admin
5. The administrator restarts pgpool

If for some reason, for example, the stopped DB has been synced with the active DB by another means, pgpool_status can be removed safely before starting pgpool.

back to top

Reloading pgpool-II configuration files

pgpool-II can reload configuration files without restarting.

```
pgpool [-c][-f config_file][-a hba_file][-F pcp_config_file] reload
```

-f config_file	--config-file config-file	specifies pgpool.conf

-a hba_file	--hba-file hba_file	specifies pool_hba.conf
-F pcp_config_file	--pcp-password-file	specifies pcp.conf

Please note that some configuration items cannot be changed by reloading. New configuration takes effect after a change for new sessions.

back to top

SHOW commands

Overview

pgpool-II provides some information via the SHOW command. SHOW is a real SQL statement, but pgPool-II intercepts this command if it asks for specific pgPool-II information. Available options are:

- pool_status, to get the configuration
- pool_nodes, to get the nodes information V3.0 -
- pool_processes, to get information on pgPool-II processes V3.0 -
- pool_pools, to get information on pgPool-II pools V3.0 -
- pool_version, to get the pgPool_II release version V3.0 -

Other than "pool_status" are added since pgpool-II 3.0.

Note : The term 'pool' refers to the pool of PostgreSQL sessions owned by one pgpool process, not the whole sessions owned by pgpool.

the "pool_status" SQL statement was already available in previous releases, but the other ones have appeared in release 3.0.

pool_status

"SHOW pool_status" sends back the list of configuration parameters with their name, value, and description. Here is an excerpt of the result:

```
benchs2=# show pool_status;
```

```
                item                |            value             |                          description
------------------------------------+------------------------------+---------------------------------------------------------
 listen_addresses                   | localhost                    | host name(s) or IP address(es) to listen to
 port                               | 9999                         | pgpool accepting port number
 socket_dir                         | /tmp                         | pgpool socket directory
 pcp_port                           | 9898                         | PCP port # to bind
 pcp_socket_dir                     | /tmp                         | PCP socket directory
```

pool_nodes V3.0 -

"SHOW pool_nodes" sends back a list of all configured nodes. It displays the node id, the hostname, the port, the status, the weight (only meaningful if you use the load balancing mode) and the role. The possible values in the status column are explained in the [pcp_node_info reference](#).

```
benchs2=# show pool_nodes;
  id  |  hostname   | port | status | lb_weight |  role
------+-------------+------+--------+-----------+---------
   0  | 127.0.0.1   | 5432 | 2      | 0.5       | primary
   1  | 192.168.1.7 | 5432 | 3      | 0.5       | standby
(2 lignes)
```

pool_processes V3.0 -

"SHOW pool_processes" sends back a list of all pgPool-II processes waiting for connections and dealing with a connection.

It has 6 columns:

- pool_pid is the PID of the displayed pgPool-II process
- start_time is the timestamp of when this process was launched
- database is the database name of the currently active backend for this process
- username is the user name used in the connection of the currently active backend for this process
- create_time is the creation time and date of the connection
- pool_counter counts the number of times this pool of connections (process) has been used by clients

This view will always return num_init_children lines.

```
benchs2=# show pool_processes;
 pool_pid |     start_time      | database | username  |     create_time     | pool_counter
----------+---------------------+----------+-----------+---------------------+--------------
```

```
 8465     | 2010-08-14 08:35:40 |          |           |                     |
 8466     | 2010-08-14 08:35:40 | benchs   | guillaume | 2010-08-14 08:35:43 | 1
 8467     | 2010-08-14 08:35:40 |          |           |                     |
 8468     | 2010-08-14 08:35:40 |          |           |                     |
 8469     | 2010-08-14 08:35:40 |          |           |                     |
(5 lines)
```

pool_pools V3.0 -

"SHOW pool_pools" sends back a list of pools handled by pgPool-II. their name, value, and description. Here is an excerpt of the result:

It has 11 columns:

- pool_pid is the PID of the pgPool-II process
- start_time is the time and date when this process was launched
- pool_id is the pool identifier (should be between 0 and max_pool-1)
- backend_id is the backend identifier (should be between 0 and the number of configured backends minus one)
- database is the database name for this process's pool id connection
- username is the user name for this process's pool id connection
- create_time is the creation time and date of this connection
- majorversion and minorversion are the version of the protocol used in this connection
- pool_counter counts the number of times this connection has been used by clients
- pool_backendpid is the PID of the PostgreSQL process
- pool_connected is a true (1) if a frontend is currently using this backend.

It'll always return num_init_children * max_pool lines.

```
 pool_pid |     start_time      | pool_id | backend_id | database | username  |     create_time     | majorversion | min
----------+---------------------+---------+------------+----------+-----------+---------------------+--------------+-----
 8465     | 2010-08-14 08:35:40 | 0       | 0          |          |           |                     |              |
 8465     | 2010-08-14 08:35:40 | 1       | 0          |          |           |                     |              |
 8465     | 2010-08-14 08:35:40 | 2       | 0          |          |           |                     |              |
 8465     | 2010-08-14 08:35:40 | 3       | 0          |          |           |                     |              |
 8466     | 2010-08-14 08:35:40 | 0       | 0          | benchs   | guillaume | 2010-08-14 08:35:43 | 3            | 0
 8466     | 2010-08-14 08:35:40 | 1       | 0          |          |           |                     |              |
 8466     | 2010-08-14 08:35:40 | 2       | 0          |          |           |                     |              |
 8466     | 2010-08-14 08:35:40 | 3       | 0          |          |           |                     |              |
 8467     | 2010-08-14 08:35:40 | 0       | 0          |          |           |                     |              |
 8467     | 2010-08-14 08:35:40 | 1       | 0          |          |           |                     |              |
 8467     | 2010-08-14 08:35:40 | 2       | 0          |          |           |                     |              |
 8467     | 2010-08-14 08:35:40 | 3       | 0          |          |           |                     |              |
```

```
 8468     | 2010-08-14 08:35:40 | 0       | 0          |          |          |                     |             |
 8468     | 2010-08-14 08:35:40 | 1       | 0          |          |          |                     |             |
 8468     | 2010-08-14 08:35:40 | 2       | 0          |          |          |                     |             |
 8468     | 2010-08-14 08:35:40 | 3       | 0          |          |          |                     |             |
 8469     | 2010-08-14 08:35:40 | 0       | 0          |          |          |                     |             |
 8469     | 2010-08-14 08:35:40 | 1       | 0          |          |          |                     |             |
 8469     | 2010-08-14 08:35:40 | 2       | 0          |          |          |                     |             |
 8469     | 2010-08-14 08:35:40 | 3       | 0          |          |          |                     |             |
(20 lines)
```

pool_version V3.0 -

"SHOW pool_version" displays a string containing the pgPool-II release number. Here is an example of it:

```
benchs2=# show pool_version;
      pool_version
-----------------------
 3.0-dev (umiyameboshi)
(1 line)
```

pool_cache V3.0 -

"SHOW pool_cache" displays cache storage statistics if on memory query cache is enabled. Here is an example of it:

```
test=# \x
\x
Expanded display is on.
test=# show pool_cache;
show pool_cache;
-[ RECORD 1 ]---------------+---------
num_cache_hits              | 891703
num_selects                 | 99995
cache_hit_ratio             | 0.90
num_hash_entries            | 131072
used_hash_entries           | 99992
num_cache_entries           | 99992
used_cache_enrties_size     | 12482600
free_cache_entries_size     | 54626264
fragment_cache_entries_size | 0
```

- num_cache_hits means the number of SELECTs which hit cache.

- num_selects means the number of SELECTs which do not hit cache.
- cache_hit_ratio means cache hit ratio, calculated from num_cache_hits/(num_cache_hits+num_selects) Anything below num_hash_entries are valid only when cache storage is on shared memory.
- num_hash_entries means number of entries in hash table, which is used for index to cache storage and should be equal to memqcache_max_num_cache in pgpool.conf. This is the upper limit for number of cache entries.
- used_hash_entries means number of already used entries in num_hash_entries.
- num_cache_entries means number of valid cache entries in the cache storage and should be equal to used_hash_entries.
- used_cache_entries_size means total size of cache storage in bytes which is already used.
- free_cache_entries_size means total size of cache storage in bytes which is not used yet or can be usable.
- fragment_cache_entries_size means total size of cache storage in bytes which cannot be used because of fragmentation.
- The fragmented area can be reused later if free_cache_entries_size becomes 0 (or there's no enough space for the SELECT result).

back to top

Online Recovery

Overview

pgpool-II, while in replication mode, can sync a database and attach a node while still servicing clients. We call this feature "online recovery".

A recovery target node must be in the detached state before doing online recovery. If you wish to add a PostgreSQL server dynamically, add 'backend_hostname' and its associated parameters and reload pgpool.conf. pgpool-II registers this new node as a detached node.

caution: Stop autovacuum on the master node (the first node which is up and running). Autovacuum may change the contents of the database and might cause inconsistency after online recovery if it's running. This applies only if you're recovering with a simple copy mechanism, such as the rsync one explained below. This doesn't apply if you're using PostgreSQL's PITR mechanism.

If the target PostgreSQL server has already started, you need to shut it down.

pgpool-II performs online recovery in two separated phases. There are a few seconds or minutes when client will be waiting to connect to pgpool-II while a recovery node synchronizes database. It follows these steps:

1. CHECKPOINT
2. First stage of online recovery
3. Wait until all clients have disconnected
4. CHECKPOINT
5. Second stage of online recovery
6. Start up postmaster (perform pgpool_remote_start)
7. Node attach

The first step of data synchronization is called "first stage". Data is synchronized during the first stage. In the first stage, data **can** be updated or retrieved from any table concurrently.

You can specify a script executed during the first stage. pgpool-II passes three arguments to the script.

1. The database cluster path of a master node.
2. The hostname of a recovery target node.
3. The database cluster path of a recovery target node.

Data synchronization is finalized during what is called "second stage". Before entering the second stage, pgpool-II waits until all clients have disconnected. It blocks any new incoming connection until the second stage is over.

After all connections have terminated, pgpool-II merges updated data between the first stage and the second stage. This is the final data synchronization step.

Note that there is a restriction about online recovery. If pgpool-II itself is installed on multiple hosts, online recovery does not work correctly, because pgpool-II has to stop all clients during the 2nd stage of online recovery. If there are several pgpool hosts, only one will have received the online recovery command and will block connections.

Configuration for online recovery

Set the following parameters for online recovery in pgpool.conf.

- backend_data_directory
- recovery_user
- recovery_password
- recovery_1st_stage_command
- recovery_2nd_stage_command

Installing C language functions

You need to install the following C language function for online recovery into the "template1" database of all backend

nodes. Its source code is in pgpool-II tarball.

```
pgpool-II-x.x.x/sql/pgpool-recovery/
```

Change directory there and do "make install".

```
% cd pgpool-II-x.x.x/sql/pgpool-recovery/
% make install
```

Then, install the SQL function.

```
% cd pgpool-II-x.x.x/sql/pgpool-recovery/
% psql -f pgpool-recovery.sql template1
```

Recovery script deployment

We must deploy some data sync scripts and a remote start script into the database cluster directory ($PGDATA). Sample script files are available in pgpool-II-x.x.x/sample directory.

Online recovery by PITR

Here is how to do online recovery by Point In Time Recovery (PITR), which is available in PostgreSQL 8.2 and later versions. Note that all PostgreSQL servers involved need to have PITR enabled.

1st stage

A script to get a base backup on a master node and copy it to a recovery target node on the first stage is needed. The script can be named "copy-base-backup" for example. Here is the sample script.

```
#! /bin/sh
DATA=$1
RECOVERY_TARGET=$2
RECOVERY_DATA=$3

psql -c "select pg_start_backup('pgpool-recovery')" postgres
echo "restore_command = 'scp $HOSTNAME:/data/archive_log/%f %p'" > /data/recovery.conf
tar -C /data -zcf pgsql.tar.gz pgsql
psql -c 'select pg_stop_backup()' postgres
```

```
scp pgsql.tar.gz $RECOVERY_TARGET:$RECOVERY_DATA
```

This script puts the master database in backup mode, generates the following recovery.conf:

```
restore_command = 'scp master:/data/archive_log/%f %p'
```

performs the backup, then puts the master database out of backup mode and copies the backup on the chosen target node.

2nd stage

The second stage of the procedure is a script to force an XLOG file switch. This script is named "pgpool_recovery_pitr" here. It enforces a switch of the transaction log. For this purpose, pg_switch_xlog could be used.

V3.1 - However it may return **before** the switch is done and this might lead to failure of the online recovery procedure. Pgpool-II provides a safer function called "pgpool_switch_xlog" which will wait until the transaction log switching is actually finished. pgpool_switch_xlog is installed during the procedure performed in the Installing C functions section.

Here is the sample script.

```
#! /bin/sh
# Online recovery 2nd stage script
#
datadir=$1       # master dabatase cluster
DEST=$2          # hostname of the DB node to be recovered
DESTDIR=$3       # database cluster of the DB node to be recovered
port=5432        # PostgreSQL port number
archdir=/data/archive_log    # archive log directory

# Force to flush current value of sequences to xlog
psql -p $port -t -c 'SELECT datname FROM pg_database WHERE NOT datistemplate AND datallowconn' template1|
while read i
do
  if [ "$i" != "" ];then
    psql -p $port -c "SELECT setval(oid, nextval(oid)) FROM pg_class WHERE relkind = 'S'" $i
  fi
done

psql -p $port -c "SELECT pgpool_switch_xlog('$archdir')" template1
```

This flushing of sequences is only useful in replication mode: in this case, sequences have to have the same starting point on all nodes. It's not useful in master-slave mode.

The loop in the script forces PostgreSQL to emit current value of all sequences in all databases in the master node to the transaction log so that it is propagated to the recovery target node.

We deploy these scripts into the $PGDATA directory.

Finally, we edit pgpool.conf.

```
recovery_1st_stage_command = 'copy-base-backup'
recovery_2nd_stage_command = 'pgpool_recovery_pitr'
```

We have finished preparing online recovery by PITR.

pgpool_remote_start

This script starts up the remote host's postmaster process. pgpool-II executes it the following way.

```
% pgpool_remote_start remote_host remote_datadir
remote_host:    Hostname of a recovery target.
remote_datadir: Database cluster path of a recovery target.
```

In this sample script, we start up the postmaster process over ssh. So you need to be able to connect over ssh without a password for it to work.

If you recover with PITR, you need to deploy a base backup. PostgreSQL will automatically start up doing a PITR recovery. Then it will accept connections.

```
#! /bin/sh
DEST=$1
DESTDIR=$2
PGCTL=/usr/local/pgsql/bin/pg_ctl

# Deploy a base backup
ssh -T $DEST 'cd /data/; tar zxf pgsql.tar.gz' 2>/dev/null 1>/dev/null < /dev/null
# Startup PostgreSQL server
ssh -T $DEST $PGCTL -w -D $DESTDIR start 2>/dev/null 1>/dev/null < /dev/null &
```

Online recovery with rsync.

PostgreSQL 7.4 does not have PITR. PostgreSQL 8.0 and 8.1 cannot force to switch transaction log. So rsync can be used to do online recovery. In the "sample" directory of pgpool-II's tarball, there is a recovery script named "pgpool_recovery". It uses the rsync command. pgpool-II calls the script with three arguments.

```
% pgpool_recovery datadir remote_host remote_datadir
datadir:        Database cluster path of a master node.
remote_host:    Hostname of a recovery target node.
```

```
remote_datadir: Database cluster path of a recovery target node.
```

This script copies physical files with rsync over ssh. So you need to be able to connect over ssh without a password.

Note about rsync:

- -c (or --checksum) option is required to enable reliable file tranmitting
- -z (or --compress) option does compression before transmitting data. This will be great for slower connection, but it might add too much CPU overhead for a 100Mbit or faster connections. In this case you might want not to use this option.
- rsync 3.0.5 has great speed performance improvements (50% faster according to a report from pgpool-general mailing list.

If you use pgpool_recovery, add the following lines into pgpool.conf.

```
recovery_1st_stage_command = 'pgpool_recovery'
recovery_2nd_stage_command = 'pgpool_recovery'
```

How to perform online recovery

In order to do online recovery, use the pcp_recovery_node command or pgpoolAdmin.

Note that you need to pass a large number to the first argument of pcp_recovery_node. It is the timeout parameter in seconds. If you use pgpoolAdmin, set "_PGPOOL2_PCP_TIMEOUT " parameter to a large number in pgmgt.conf.php.

PostgreSQL version up using online recovery

replication mode case

You can update PostgreSQL on each node without stopping pgpool-II if pgpool-II operated in replication mode. Please note that active sessions from clients to pgpool-II will be disconnected while disconnecting and attaching DB nodes. Also please note that you cannot do major version up in the method described below (i.e. the version up should not require dump/restore).

1. Prepare online recovery.
2. Version up should perform nodes which are not master node first. Stop PostgreSQL on a non-master node. Pgpool-II will detect PostgreSQL termination and degenerate emitting logs

below. At this point all sessions connected to pgpool-II disconnected.

```
2010-07-27 16:32:29 LOG:   pid 10215: set 1 th backend down status
2010-07-27 16:32:29 LOG:   pid 10215: starting degeneration. shutdown host localhost(5433)
2010-07-27 16:32:29 LOG:   pid 10215: failover_handler: set new master node: 0
2010-07-27 16:32:29 LOG:   pid 10215: failover done. shutdown host localhost(5433)
```

3. Version up PostgreSQL on the stopping node. You can overwrite old PostgreSQL, we recommend move old PostgreSQL somewhere so that you could recover it just in case however.
4. If you install new PostgreSQL in different location from the old one and do not want to update your recovery script, you need to match the path by using tools including symbolic link. If you choose to overwrite, you can skip following steps till installation of C function step. You can execute online recovery immediately.
5. Change installation directory of old PostgreSQL. Installting directory of PostgreSQL is supposed to be /usr/local/pgsql in following description.

```
$ mv /usr/local/pgsql /usr/local/pgsql-old
```

6. Create a symbolic link to the location where newer version of PostgreSQL installed. This allow you to continue to use command search path you currently use. Installing directory of newer PostgreSQL is supposed to be /usr/local/pgsql-new in following description.

```
$ ln -s /usr/local/pgsql-new /usr/local/pgsql
```

7. If database directory is located under older PostgreSQL installation directory, you should create or copy so that newer PostgreSQL can access it. We use symbolic link in the following example.

```
$ ln -s /usr/local/pgsql-old/data /usr/local/pgsql/data
```

8. Install C functions into PostgreSQL. "Installing C functions" section may help you. Because online recovery copies database cluster, the last step installing functions using psql is not necessary. Do make install.
9. Do online recovery. You are done with one node version up. To execute online recovery, you can use pcp_recovery_node or pgpoolAdmin.
10. Repeat steps above on each node. In the very last master node should be updated. You are done.

If you are using streaming replication

You can update standby PostgreSQL server without stopping pgpool-II.

The procedure to update standby PostgreSQL servers are same as the one of replication mode. Please refer to "Online recovery with Streaming Replication" to set up recovery_1st_stage_command and recovery_2nd_stage_command.

You cannot version up primary server without stopping pgpool-II. You need to stop pgpool-II while updating primary server. The procedure to update primary PostgreSQL server is same as the one standby server. The procedure to update primary PostgreSQL server is as follows:

1. Stop pgpool-II
2. Stop primary PostgreSQL
3. Update primary PostgreSQL
4. Start primary PostgreSQL
5. Start pgpool-II

back to top

Backup

To back up backend PostgreSQL servers and system DB, you can use physical backup, logical backup (pg_dump, pg_dumpall) and PITR in the same manner as PostgreSQL. Please note that using logical backup and PITR should be performed directory with PostgreSQL, rather than via pgpool-II to avoid errors caused by load_balance_mode and replicate_select.

replication mode and master/slave mode

If pgpool-II is operated in replication mode or master/slave mode, take a backup on one DB nodes in the cluster.

If you are using master/slave mode and asynchronous replication systems(Slony-I and streaming replication) and need the latest backup, you should take a backup on the master node.

pg_dump takes ACCESS SHARE lock on database. Commands taking ACCESS EXECUTE lock, such as ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER and VACUUM FULL will wait for the completion of pg_dump because of lock conflict. Also this may affect the primary node even if you are doing pg_dump on standby.

Parallel mode

If you are using parallel mode and need to take a consistent backup, you need to stop pgpool-II.

To use logical backup, stop applications and pgpool-II then perform pg_dump or pg_dumpall on all nodes. After finishing backup, start pgpool-II then start applications.

To use PITR, please make sure that system times are identical on all nodes. Prepare archive logging and take base backup. After finishing backup, stop and restart applications or pgpool-II. Record the time of stop and start. This temporary stop will make consistent state among all over cluster. If you need to restore form the base backup and archive log, set recovery_target_time of recovery.conf in the middle of the start/stop time.

Backing up of system DB

You need to backup system DB if pgpool-II is operated in parallel query mode or if you are using query cache. Backup database specified by system_db_dbname in pgpool.conf.

back to top

Deploying pgpool-II

pgpool-II can run on a dedicated server, on the server where application server is running on or other servers. In this section we discuss how to make those deployments and pros and cons.

Dedicated server

Pgpool-II is running on a dedicated server. It's simple and pgpool-II is not affected by other server softwares. Obvious cons is you need to buy more hardware. Also pgpool-II can be a single point of failure with this configuration (you can avoid this by enabling watchdog or using pgpool-HA described described below).

Deploying on a web server or application server

Deploying pgpool-II on a server where Apache, JBoss, Tomcat or other web server and application servers. Since communication between pgpool-II and web servers and application servers is within a local machine, socket communication can be faster than inter-sever communication. Also if you are using multiple web serves or application servers, you can avoid the single point of failure problem (in this case you must have identical pgpool.conf on each pgpool-II instance except the watchdog section).

We strongly recommend to enable watchdog to avoid following concerns in this configuration.

- If the communication between pgpool-II and DB servers is not stable, it is possible that DB node #1 is down from a point of a pgpool-II instance, while it is up from other pgpool-II instance's point of view. To avoid this, you can multiplex the network connection.
- While executing online recovery in replication mode, you need to stop all pgpool-II instances

except the one which is doing online recovery. Otherwise DB can be running into inconsistent state. In master slave mode+streaming replication mode, you do not need to stop other pgpool-II instances. You should not execute online recovery at the same time on multiple pgpool-II instances however.

Running pgpool-II on DB server

Running pgpool-II on the server as PostgreSQL is running on. You can avoid the single point of failure problem of pgpool-II with configuration. And obviously you do need to buy additional dedicated server. Problem with this configuration is, application need to aware of which DB server they should connect to. To solve the problem you can use virtual IP with watchdog or pgpool-HA.

About pgpool-HA

Pgpool-HA is a high availability software for pgpool-II using heartbeat. Pgpool-HA is a sub-project of the pgpool project as well as pgpool-II. Pgpool-HA can be available from the pgpool development site as an open source software.

back to top

Watchdog V3.2 -

What is watchdog

"Watchdog" is a sub process of pgpool-II to add high availability. This resolve the single point of failure by coordinating multiple pgpool-IIs. watchdog adds the following features to pgpool-II.

Life checking of pgpool-II

Watchdog monitors pgpool-IIs by either of two methods, "heartbeat" mode or "query" mode.

- In heartbeat mode, watchdog monitors other pgpool-II processes by using heartbeat signal. Watchdog receives heartbeat signals sent by other pgpool-II periodically. If there are no signal for a certain period, watchdog regards this as failure of the pgpool-II. For redundancy you can use multiple network connections for heartbeat exchange between pgpool-IIs. This is the default mode and recommended.

- In query mode, watchdog monitors pgpool-II's service rather than process. watchdog sends queries to other pgpool-II and checks the response. Note that this method requires connections from other pgpool-IIs, so it would fail motoring if num_init_children isn't large enough. This mode is deprecated and left for backward compatibility.

Also watchdog monitors connections to upstream servers (application servers etc.) from the pgpool-II, and checks whether the pgpool-II can serves to the servers. If the monitoring fails, watchdog treats the pgpool-II as down.

Coordinating multiple pgpool-IIs

Watchdog coordinates multiple pgpool-IIs by exchanging information with each other.

- When backend node status changes by failover etc., watchdog notifies the information to other pgpool-IIs and synchronizes them. When online recovery occurs, watchdog restricts client connections to other pgpool-IIs for avoiding inconsistency between backends.
- Commands on failback or failover (failover_command, failback_command, follow_master_command) are executed by only one pgpool-II by interlocking.

Changing active/standby state in case of certain faults detected

When a fault of pgpool-II is detected, watchdog notifies the other watchdogs of it. If this is the active pgpool-II, watchdogs decides the new active pgpool-II by voting and change active/standby state.

Automatic virtual IP address assigning synchronous to server switching

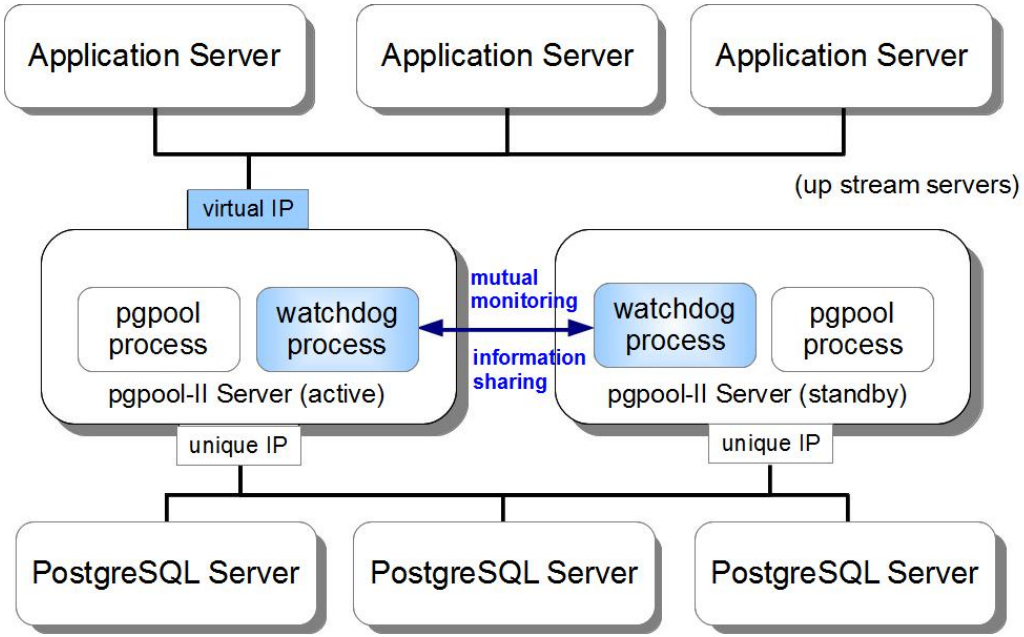
When a standby pgpool-II server promotes to active, the new active server brings up virtual IP interface. Meanwhile, the previous active server brings down the virtual IP interface. This enables the active pgpool-II to work using the same IP address even when servers are switched.

Automatic registration of a server as standby in recovery

When broken server recovers or new server is attached, the watchdog process notifies the other watchdog process along with information of the new server, and the watchdog process receives information on the active server and other servers. Then, the attached server is registered as a standby.

Server composition

Figure below describes how pgpool-II and watchdog process is configured.



Starting/stopping watchdog

Watchdog process starts/stops automatically as sub-processes of pgpool-II, therefore there is no dedicated command to start/stop it.

Watchdog requires root privilege for controling the virtual IP interface. One method is to start pgpool-II by root privilege. However, for security reason, to set custom commands to if_up_cmd, if_up_cmd, if_up_cmd using sudo or setuid is recommended method.

Watchdog's life-checking starts after after all of the pgpool-IIs has started. This doesn't start if not all "other" nodes are alive Until this, failover of the virtual IP never occurs.

Configuring watchdog (pgpool.conf)

Watchdog configuration parameters are described in pgpool.conf. There is sample configuration in WATCHDOG section in pgpool.conf.sample file.

All following options are required to be specified in watchdog process.

Enabling

use_watchdog V3.2 -

If on, activates watchdog. Default is off.

You need to restart pgpool-II if you change this value.

Watchdog communication

wd_hostname V3.2 -

Specifies the hostname or IP address of pgpool-II server. This is used for sending/receiving queries and packets, and also as identifier of watchdog.

You need to restart pgpool-II if you change this value.

wd_port V3.2 -

Specifies the port number for watchdog communication.

You need to restart pgpool-II if you change this value.

wd_authkey V3.3 -

This option specifies the authentication key used in watchdog communication. All the pgpool-II must have the same key. Packets from watchdog of wrong key will be rejects. This authentication is applied also for heatrbeat singals if lifecheck method is heartbeat mode. If this is empty (default), watchdog doesn't conduct authenticate.

You need to restart pgpool-II if you change this value.

Connection to up stream servers

trusted_servers V3.2 -

The list of trusted servers to check the up stream connections. Each server is required to respond to ping. Specify a comma separated list of servers such as "hostA,hostB,hostC". If none of the server are pingable, watchdog regards it as failure of the pgpool-II.

If this option is empty, watchdog doesn't check up stream connections.

You need to restart pgpool-II if you change this value.

ping_path V3.2 -

This parameter specifies a path of ping command for monitoring connection to the upper servers. Set the only path such as "/bin".

You need to restart pgpool-II if you change this value.

Virtual IP control

Configuration about virtual IP interface control

delegate_IP V3.2 -

Specifies the virtual IP address (VIP) of pgpool-II that is connected from client servers (application servers etc.). When a pgpool-II is switched from standby to active, the pgpool-II takes over this VIP. If this option is emply, virtual IP is never brought up.

You need to restart pgpool-II if you change this value.

ifconfig_path V3.2 -

This parameter specifies a path of a command to switch the IP address. Set the only path such as "/sbin".

You need to restart pgpool-II if you change this value.

if_up_cmd V3.2 -

This parameter specifies a command to bring up the virtual IP. Set the command and parameters such as "ifconfig eth0:0 inet $_IP_$ netmask 255.255.255.0". $_IP_$ is replaced by the IP address specified in delegate_IP.

You need to restart pgpool-II if you change this value.

if_down_cmd V3.2 -

This parameter specifies a command to bring down the virtual IP. Set the command and parameters such as "ifconfig eth0:0 down".

You need to restart pgpool-II if you change this value.

arping_path V3.2 -

This parameter specifies a path of a command to send an ARP request after the virtual IP is switched. Set the only path such as "/usr/sbin".

You need to restart pgpool-II if you change this value.

arping_cmd V3.2 -

This parameter specifies a command to send an ARP request after the virtual IP is switched. Set the command and parameters such as "arping -U $_IP_$ -w 1". $_IP_$ is replaced by the IP address specified in delegate_IP.

You need to restart pgpool-II if you change this value.

Behaivor on escalation

Configuration about behavior when pgpool-II escalates to active (virtual IP holder)

clear_memqcache_on_escalation V3.3 -

If this is on, watchdog clears all the query cache in the shared memory when pgpool-II escaltes to active. This prevents the new active pgpool-II from using old query caches inconsistence to the old active. Default is on.

This works only if memqcache_method is 'shmem'.

wd_escalation_command V3.3 -

Watchdog executes this command on the new active when pgpool-II escaltes to active. The timing is just before virtual IP brings up.

Life checking pgpool-II

Watchdog checks pgpool-II status periodically. This is called "life check".

Common

wd_lifecheck_method V3.3 -

This parameter specifies the method of life check. This is either of 'heartbeat' (default) or 'query'.

In 'heartbeat' mode, watchdog sends heartbeat singals (UDP packets) periodically to other pgpool-II. Watchdog also receives the signals from other pgpool-II. If there are no signal for a certain period, watchdog regards is as failure of the pgpool-II.

In 'query' mode, watchdog sends monitoring queries to other pgpool-II and checks the response.

CAUTION: In query mode, you need to set num_init_children large enough if you plan to use watchdog. This is because the watchdog process connects to pgpool-II as a client.

You need to restart pgpool-II if you change this value.

wd_interval V3.2 -

This parameter specifies the interval between life checks of pgpool-II in second. (A number greater than or

equal to 1) Default is 10.

You need to restart pgpool-II if you change this value.

Configuration of heartbeat mode

wd_heartbeat_port V3.3 -

This option specifies the port number to receive heartbeat signals. This works only heartbeat mode.

You need to restart pgpool-II if you change this value.

wd_heartbeat_keepalive V3.3 -

This option specifies the interval time (sec.) of sending heartbeat signals. Default is 2. This works only heartbeat mode.

You need to restart pgpool-II if you change this value.

wd_heartbeat_deadtime V3.3 -

If there are no heartbeat signal for the period specified by this option, watchdog regards it as failure of the remote pgpool-II. This works only heartbeat mode.

You need to restart pgpool-II if you change this value.

heartbeat_destination0 V3.3 -

This option specifies the destination of heartbeat signals by IP address or hostname. You can use multiple destination. The number at the end of the parameter name is referred as "destination number", and it starts from 0. This works only heartbeat mode.

You need to restart pgpool-II if you change this value.

heartbeat_destination_port0 V3.3 -

This option specifies the port number of destination of heartbeat signals which is specified by heartbeat_destinationX. This is usually the same value as wd_heartbeat_port You must use another value if the port number is unusable on a certain host or there are more than two pgpool-IIs in a host. The number at the end of the parameter name is referred as "destination number", and it starts from 0. This works only heartbeat mode.

You need to restart pgpool-II if you change this value.

heartbeat_device0 V3.3 -

This option specifies the network device name for sending heartbeat signals to destination specified by heartbeat_destinationX. You can use the same device for different distinations. The number at the end of the parameter name is referred as "destination number", and it starts from 0. This works only heartbeat mode. This is ignored when the value is empty. In addition, this works only when pgpool-II has root privilege and are running on Linux, because this uses SO_BINDTODEVICE socket option.

You need to restart pgpool-II if you change this value.

Configuration of query mode

wd_life_point V3.2 -

The times to retry a failed life check of pgpool-II. (A number greater than or equal to 1) Default is 3. This works only query mode.

You need to restart pgpool-II if you change this value.

wd_lifecheck_query V3.2 -

Actual query to check pgpool-II. Default is "SELECT 1". This works only query mode.

You need to restart pgpool-II if you change this value.

wd_lifecheck_dbname V3.3 -

The database name connected for checking pgpool-II. Default is "template1". This works only query mode.

wd_lifecheck_user V3.3 -

The user name to check pgpool-II. This user must exist in all the PostgreSQL backends. Default is "nobody". This works only query mode.

wd_lifecheck_password V3.3 -

The password of the user to check pgpool-II. Default is "". This works only query mode.

Servers to monitor

other_pgpool_hostname0 V3.2 -

Specifies the hostname pgpool-II server to be monitored. This is used for sending/receiving queries and packets, and also as identifier of watchdog. The number at the end of the parameter name is referred as "server id", and it starts from 0.

You need to restart pgpool-II if you change this value.

other_pgpool_port0 V3.2 -

Specifies the port number for pgpool service of pgpool-II server to be monitored. In query mode, the queries specified in wd_lifecheck_query is sent to this port. The number at the end of the parameter name is referred as "server id", and it starts from 0.

You need to restart pgpool-II if you change this value.

other_wd_port0 V3.2 -

Specifies the port number for watchdog on pgpool-II server to be monitored. The number at the end of the parameter name is referred as "server id", and it starts from 0.

You need to restart pgpool-II if you change this value.

Restrictions on watchdog

- In query mode, when all the DB nodes are detached from a pgpool-II due to PostgreSQL server failure or pcp_detach_node issued, watchdog regards that the pgpool-II service is in down status and brings the virtual IP assigned to watchdog down. Thus clients of pgpool cannot connect to pgpool-II using the virtual IP any more. This is neccessary to avoid *split-brain*, that is, situations where there are multiple active pgpool-II.
- Don't connect to pgpool-II in down status using the real IP. Because a pgpool-II in down status can't receive information from watchdog, its backend status may be different from other pgpool-II.
- pgpool-II in down status can't become active nor standby pgpool-II. Recovery from down status requires restart of pgpool-II.
- Please note that after the active pgpool-II stops, it will take a few seconds until the standby pgpool-II promote to new active, to make sure that the former virtual IP is brought down before a down notification packet is sent to other pgpool-IIs.

back to top

PCP Commands

PCP commands list

PCP commands are UNIX commands which manipulate pgpool-II via the network.

pcp_node_count	retrieves the number of nodes
pcp_node_info	retrieves the node information
pcp_watchdog_info V3.3 -	retrieves the watchdog information
pcp_proc_count	retrieves the process list
pcp_proc_info	retrieves the process information

pcp_pool_status V3.1 -	retrieves parameters in pgpool.conf
pcp_systemdb_info	retrieves the System DB information
pcp_detach_node	detaches a node from pgpool-II
pcp_attach_node	attaches a node to pgpool-II
pcp_promote_node V3.1 -	promote a new master node to pgpool-II
pcp_stop_pgpool	stops pgpool-II

Common Command-line Arguments

There are five arguments common to all of the PCP commands. They give information about pgpool-II and authentication. Extra arguments may be needed for some commands.

```
e.g.)  $ pcp_node_count 10 localhost 9898 postgres hogehoge
```

First argument	timeout value in seconds. PCP disconnects if pgpool-II does not respond in this many seconds.
Second argument	pgpool-II's hostname
Third argument	PCP port number
Fourth argument	PCP user name
Fifth argument	PCP password

PCP user names and passwords must be declared in `pcp.conf` in `$prefix/etc` directory. `-F` option can be used when starting pgpool-II if `pcp.conf` is placed somewhere else. The password does not need to be in md5 format when passing it to the PCP commands.

PCP Commands to know status and configuration

All PCP commands display the results to the standard output.

pcp_node_count

Format `pcp_node_count _timeout_ _host_ _port_ _userid_ _passwd_`

Desc. Displays the total number of nodes defined in `pgpool.conf`. It does not distinguish between nodes status, ie attached/detached. ALL nodes are counted.

pcp_node_info

Format `pcp_node_info _timeout_ _host_ _port_ _userid_ _passwd_ _nodeid_`

Desc. Displays the information on the given node ID. Here is an output example:

```
$ pcp_node_info 10 localhost 9898 postgres hogehoge 0
host1 5432 1 1073741823.500000
```

The result is in the following order:

- 1. hostname
- 2. port number
- 3. status
- 4. load balance weight

Status is represented by a digit from [0 to 3].

- 0 - This state is only used during the initialization. PCP will never display it.
- 1 - Node is up. No connections yet.
- 2 - Node is up. Connections are pooled.
- 3 - Node is down.

The load balance weight is displayed in normalized format.

The `--verbose` option can help understand the output. For example:

```
$ pcp_node_info --verbose 10 localhost 9898 postgres hogehoge 0
Hostname: host1
Port    : 5432
```

```
Status  : 1
Weight  : 0.5
```

Specifying an invalid node ID will result in an error with exit status 12, and BackendError will be displayed.

pcp_watchdog_info V3.3 -

Format `pcp_watchdog_info _timeout_ _host_ _port_ _userid_ _passwd_ [_watchdogid_]`

Desc. Displays the watchdog status of the pgpool-II. _watchdogid_ is the index of other_pgpool_hostname parameter in pgpool.conf.

If this is omitted, display the watchdog status of the pgpool-II specified by _host_:_port_.

Here is an output example:

```
$ pcp_watchdog_info 10 localhost 9898 postgres hogehoge 0
host1 9999 9000 2
Here is an output example:
```

The result is in the following order:

- 1. hostname
- 2. port number for pgpool-II
- 3. port number for watchdog
- 4. watchdog Status

Status is represented by a digit from [1 to 4].

- 1 - watchdog is not started
- 2 - Standby: not holding the virtual IP
- 3 - Active: holding the virtual IP
- 4 - Down

The load balance weight is displayed in normalized format.

Specifying an invalid watchdog ID will result in an error with exit tatus 12, and BackendError will be displayed.

pcp_proc_count

Format `pcp_proc_count _timeout_ _host_ _port_ _userid_ _passwd_`

Desc. Displays the list of pgpool-II children process IDs. If there is more than one process, IDs will be delimited by a

white space.

pcp_proc_info

Format `pcp_proc_info _timeout_ _host_ _port_ _userid_ _passwd_ _processid_`

Desc. Displays the information on the given pgpool-II child process ID. The output example is as follows:

```
$ pcp_proc_info 10 localhost 9898 postgres hogehoge 3815
postgres_db postgres 1150769932 1150767351 3 0 1 1467 1
postgres_db postgres 1150769932 1150767351 3 0 1 1468 1
```

The result is in the following order:

- 1. connected database name
- 2. connected user name
- 3. process start-up timestamp
- 4. connection created timestamp
- 5. protocol major version
- 6. protocol minor version
- 7. connection-reuse counter
- 8. PostgreSQL backend process id
- 9. 1 if frontend conncted 0 if not

If there is no connection to the backends, nothing will be displayed. If there are multiple connections, one connection's information will be displayed on each line multiple times. Timestamps are displayed in EPOCH format.

The --verbose option can help understand the output. For example:

```
$ pcp_proc_info --verbose 10 localhost 9898 postgres hogehoge 3815
Database     : postgres_db
Username     : postgres
Start time   : 1150769932
Creation time: 1150767351
Major        : 3
Minor        : 0
Counter      : 1
PID          : 1467
Connected    : 1
Database     : postgres_db
Username     : postgres
Start time   : 1150769932
Creation time: 1150767351
```

```
Major        : 3
Minor        : 0
Counter      : 1
PID          : 1468
Connected    : 1
```

Specifying an invalid node ID will result in an error with exit status 12, and BackendError will be displayed.

pcp_pool_status V3.1 -

Format `pcp_pool_status _timeout_ _host_ _port_ _userid_ _passwd_`

Desc. Displays parameters in pgpool.conf. The output example is as follows:

```
$ pcp_pool_status 10 localhost 9898 postgres hogehoge
name : listen_addresses
value: localhost
desc : host name(s) or IP address(es) to listen to

name : port
value: 9999
desc : pgpool accepting port number

name : socket_dir
value: /tmp
desc : pgpool socket directory

name : pcp_port
value: 9898
desc : PCP port # to bind
```

pcp_systemdb_info

Format `pcp_systemdb_info _timeout_ _host_ _port_ _userid_ _passwd_`

Desc. Displays the System DB information. The output example is as follows:

```
$ pcp_systemdb_info 10 localhost 9898 postgres hogehoge
localhost 5432 yamaguti '' pgpool_catalog pgpool 3
yamaguti public accounts aid 4 aid bid abalance filler integer integer integer character(84) dist_def_accour
yamaguti public branches bid 3 bid bbalance filler integer integer character(84) dist_def_branches
yamaguti public tellers bid 4 tid bid tbalance filler integer integer integer character(84) dist_def_tellers
```

First, the System DB information will be displayed on the first line. The result is in the following order:

- 1. hostname
- 2. port number
- 3. user name
- 4. password. '' for no password.
- 5. schema name
- 6. database name
- 7. number of partioning rules defined

Second, partitioning rules will be displayed on the following lines. If there are multiple definitions, one definition will be displayed on each line multiple times. The result is in the following order:

- 1. targeted partitioning database name
- 2. targeted partitioning schema name
- 3. targeted partitioning table name
- 4. partitioning key column name
- 5. number of columns in the targeted table
- 6. column names (displayed as many as said in 5.)
- 7. column types (displayed as many as said in 5.)
- 8. partitioning rule function name

If the System DB is not defined (i.e. not in pgpool-II mode, and query cache is disabled), it results in error with exit status 12, and BackendError will be displayed.

PCP Commands to control backend nodes, etc.

pcp_detach_node

Format `pcp_detach_node [-g] _timeout_ _host_ _port_ _userid_ _passwd_ _nodeid_`
Desc.

Detaches the given node from pgpool-II. If -g is given, wait until all clients are disconnected (unless client_idle_limit_in_recovery is -1 or recovery_timeout is expired).

pcp_attach_node

Format `pcp_attach_node _timeout_ _host_ _port_ _userid_ _passwd_ _nodeid_`

Desc. Attaches the given node to pgpool-II.

pcp_promote_node V3.1 -

Format `pcp_promote_node [-g] _timeout_ _host_ _port_ _userid_ _passwd_ _nodeid_`

Desc. Promotes the given node as new master to pgpool-II. In master/slave streaming replication only. If -g is given, wait until all clients are disconnected (unless client_idle_limit_in_recovery is -1 or recovery_timeout is expired).

pcp_stop_pgpool

Format: `pcp_stop_pgpool _timeout_ _host_ _port_ _userid_ _passwd_ _mode_`

Desc. Terminate the pgpool-II process with the given shutdown mode. The available modes are as follows:

- s - smart mode
- f - fast mode
- i - immediate mode

If the pgpool-II process does not exist, it results in error with exit status 8, and ConnectionError will be displayed.

* Currently, there is no difference between the fast and immediate mode. pgpool-II terminates all the processes whether there are clients connected to the backends or not.

pcp_recovery_node

Formet `pcp_recovery_node _timeout_ _host_ _port_ _userid_ _passwd_ _nodeid_`

Desc. Attaches the given backend node with recovery.

Exit Status

PCP commands exits with status 0 when everything goes well. If any error had occurred, it will exit with the following error status.

UNKNOWNERR	1	Unknown Error (should not occur)
EOFERR	2	EOF Error
NOMEMERR	3	Memory shortage
READERR	4	Error while reading from the server
WRITEERR	5	Error while writing to the server
TIMEOUTERR	6	Timeout
INVALERR	7	Argument(s) to the PCP command was invalid
CONNERR	8	Server connection error
NOCONNERR	9	No connection exists
SOCKERR	10	Socket error
HOSTERR	11	Hostname resolution error
BACKENDERR	12	PCP process error on the server (specifying an invalid ID, etc.)
AUTHERR	13	Authorization failure

back to top

Troubleshooting

This section describes problems and their workarounds while you are using pgpool-II.

Health check failed

Pgpool-II's health checking feature detects DB nodes failure.

```
2010-07-23 16:42:57 ERROR: pid 20031: health check failed. 1 th host foo at port 5432 is down
2010-07-23 16:42:57 LOG:   pid 20031: set 1 th backend down status
2010-07-23 16:42:57 LOG:   pid 20031: starting degeneration. shutdown host foo(5432)
2010-07-23 16:42:58 LOG:   pid 20031: failover_handler: set new master node: 0
2010-07-23 16:42:58 LOG:   pid 20031: failover done. shutdown host foo(5432)
```

The log shows that the DB node 1 (host foo) goes down and disconnected (shutdown) from pgpool, and then that DB node 0 becomes new master. Check DB node 1 and remove the cause of failure. After that perform an online recovery against DB node 1 if possible.

Failed to read kind from frontend

```
2010-07-26 18:43:24 LOG:   pid 24161: ProcessFrontendResponse: failed to read kind from frontend. frontend ab
```

This log indicates that the frontend program didn't disconnect properly from pgpool-II. The possible causes are: bugs of client applications, forced termination (kill) of a client application, or temporary network failure. This kind of events don't lead to a DB destruction or data consistency problem. It's only a warning about a protocol violation. It is advised that you check the applications and networks if the message keeps on occurring.

Kind mismatch errors

It is possible that you get this error when pgpool-II operates in replication mode.

```
2010-07-22 14:18:32 ERROR: pid 9966: kind mismatch among backends. Possible last query was: "FETCH ALL FROM c
```

Pgpool-II waits for responses from the DB nodes after sending an SQL command to them. This message indicates that not all DB nodes returned the same kind of response. You'll get the SQL statement which possibly caused the error after "Possible last query was:". Then the kind of response follows. If the response indicates an error, the error message from PostgreSQL is shown. Here you see "0[T]" displaying the DB node responses: "0[T]" (starting to send row description), and "1[E" indicates that DB node 1 returns an error with message "cursor "c" does not exist", while DB node 0 sends a row description.

Caution: You will see this error when operating in master/slave mode as well. For example, even in the master/slave mode, SET command will be basically sent to all DB nodes to keep all the DB nodes in the same state.

Check the databases and re-sync them using online recovery if you find that they are out of sync.

Pgpool detected difference of the number of inserted, updated or deleted tuples

In replication mode, pgpool-II detects a different number of INSERT/UPDATE/DELETE rows on affected

nodes.

```
2010-07-22 11:49:28 ERROR: pid 30710: pgpool detected difference of the number of inserted, updated or delete
2010-07-22 11:49:28 LOG:   pid 30710: ReadyForQuery: Degenerate backends: 1
2010-07-22 11:49:28 LOG:   pid 30710: ReadyForQuery: Affected tuples are: 0 1
```

In the example above, the returned number of updated rows by "update t1 set i = 1" was different among DB nodes. The next line indicates that DB 1 got degenerated (disconnected) as a consequence, and that the number of affected rows for DB node 0 was 0, while for DB node 1 that was 1.

Stop the DB node which is suspected of having wrong data and do an online recovery.

back to top

Restrictions

Functionality of PostgreSQL

- If you use pg_terminate_backend() to stop a backend, this will trigger a failover. The reason why this happens is that PostgreSQL sends exactly the same message for a terminated backend as for a full postmaster shutdown. There is no workaround as of today. Please do not use this function.

Authentication / Access Controls

- In the replication mode or master/slave mode, trust, clear text password, and pam methods are supported. md5 in also supported since pgpool-II 3.0. md5 is supported by using a authencitaion file ("pool_passwd"). "pool_passwd" is default name of the authentication file. You can change the file name using pool_passwd. Here are the steps to enable md5 authentication:
    1. Login as the database's operating system user and type "pg_md5 --md5auth --username= " user name and md5 encrypted password are registered into pool_passwd. If pool_passwd does not exist yet, pg_md5 command will automatically create it for you.
    2. The format of pool_passwd is "username:encrypted_passwd".
    3. You also need to add an appropriate md5 entry to pool_hba.conf. See Setting up pool_hba.conf for client authentication (HBA) for more details.

4. Please note that the user name and password must be identical to those registered in PostgreSQL.
5. After changing md5 password(in both pool_passwd and PostgreSQL of course), you need to execute "pgpool reload" or restart pgpool(if your pgpool is 3.1 or before).

- In all the other modes, trust, clear text password, crypt, md5, and pam methods are supported.
- pgpool-II does not support pg_hba.conf-like access controls. If TCP/IP connections are enabled, pgpool-II accepts all the connections from any host. If needed, use iptables and such to control access from other hosts. (PostgreSQL server accepting pgpool-II connections can use pg_hba.conf, of course).

Large objects

pgpool-II 2.3.2 or later supports large object replication if the backend is PostgreSQL 8.1 or later. For this, you need to enable lobj_lock_table directive in pgpool.conf. Large object replication using backend function lo_import is not supported, however.

Temporary tables in master/slave mode

Creating/inserting/updating/deleting temporary tables are always executed on the master(primary). With pgpool-II 3.0 or later, SELECT on these tables is executed on master as well. However if the temporary table name is used as a literal in SELECT, there's no way to detect it, and the SELECT will be load balanced. That will trigger a "not found the table" error or will find another table having same name. To avoid the problem, use /*NO LOAD BALANCE*/ SQL comment.

```
Sample SELECT which causes a problem:
SELECT 't1'::regclass::oid;
```

psql's \d command uses literal table names. pgpool-II 3.0 or later checks if the SELECT includes any access to system catalogs and always send these queries to the master. Thus we avoid the problem.

Functions, etc. In Replication Mode

There is no guarantee that any data provided using a context-dependent mechanism (e.g. random number, transaction ID, OID, SERIAL, sequence), will be replicated correctly on multiple backends.

For SERIAL, enabling insert_lock will help replicating data. insert_lock also helps SELECT setval() and SELECT nextval().

In pgpool-II 2.3 or later, INSERT/UPDATE using CURRENT_TIMESTAMP, CURRENT_DATE, now() will be replicated

correctly. INSERT/UPDATE for tables using CURRENT_TIMESTAMP, CURRENT_DATE, now() as their DEFAULT values will also be replicated correctly. This is done by replacing those functions by constants fetched from master at query execution time. There are a few limitations however:

- In pgpool-II 3.0 or before, the calculation of temporal data in table default value is not accurate in some cases. For example, the following table definition:

```
CREATE TABLE rel1(
  d1 date DEFAULT CURRENT_DATE + 1
)
```

  is treated the same as:

```
CREATE TABLE rel1(
  d1 date DEFAULT CURRENT_DATE
)
```

  pgpool-II 3.1 or later handles these cases correctly. Thus the column "d1" will have tomorrow as the default value. However this enhancement does not apply if extended protocols(used in JDBC, PHP PDO for example) or PREPARE are used.

  Please note that if the column type is not a temporal one, rewriting is not performed. Such example:

```
foo bigint default (date_part('epoch'::text,('now'::text)::timestamp(3) with time zone) * (1000)::
```

- Suppose we have the following table:

```
CREATE TABLE rel1(
  c1 int,
  c2 timestamp default now()
)
```

  We can replicate

```
INSERT INTO rel1(c1) VALUES(1)
```

  since this turn into

```
INSERT INTO rel1(c1, c2) VALUES(1, '2009-01-01 23:59:59.123456+09')
```

  However,

```
INSERT INTO rel1(c1) SELECT 1
```

> cannot to be transformed, thus cannot be properly replicated in the current implementation. Values will still be inserted, with no transformation at all.

Tables created by `CREATE TEMP TABLE` will be deleted at the end of the session by specifying DISCARD ALL in reset_query_list if you are using PostgreSQL 8.3 or later.

For 8.2.x or earlier, `CREATE TEMP TABLE` will not be deleted after exiting a session. It is because of the connection pooling which, from PostgreSQL's backend point of view, keeps the session alive. To avoid this, you must explicitly drop the temporary tables by issuing `DROP TABLE`, or use `CREATE TEMP TABLE ... ON COMMIT DROP` inside the transaction block.

Queries

Here are the queries which cannot be processed by pgpool-II

INSERT (for parallel mode)

You cannot use `DEFAULT` with the partitioning key column. For example, if the column x in the table t was the partitioning key column,

```
INSERT INTO t(x) VALUES (DEFAULT);
```

is invalid. Also, functions cannot be used for this value either.

```
INSERT INTO t(x) VALUES (func());
```

Constant values must be used to INSERT with the partitioning key. `SELECT INTO` and `INSERT INTO ... SELECT` are also not supported.

UPDATE (for parallel mode)

Data consistency between the backends may be lost if the partitioning key column values are updated. pgpool-II does not re-partition the updated data.

A transaction cannot be rolled back if a query has caused an error on some backends due to a constraint violation.

If a function is called in the `WHERE` clause, that query might not be executed correctly, for example:

```
UPDATE branches set bid = 100 where bid = (select max(bid) from beances);
```

SELECT ... FOR UPDATE (for parallel mode)

If a function is called in the `WHERE` clause, that query might not be executed correctly. For example:

```
SELECT * FROM  branches where bid = (select max(bid) from beances) FOR UPDATE;
```

COPY (for parallel mode)

`COPY BINARY` is not supported. Copying from files is also not supported. Only `COPY FROM STDIN` and `COPY TO STDOUT` are supported.

ALTER/CREATE TABLE (for parallel mode)

To update the partitioning rule, pgpool-II must be restarted in order to read them from the System DB.

Transaction (for parallel mode)

`SELECT` statements executed inside a transaction block will be executed in a separate transaction. Here is an example:

```
BEGIN;
INSERT INTO t(a) VALUES (1);
SELECT * FROM t ORDER BY a; <-- INSERT above is not visible from this SELECT statement
END;
```

Views / Rules (for parallel mode)

The same definition will be created on all the backends for views and rules.

```
SELECT * FROM a, b where a.i = b.i
```

JOINs like above will be executed on each backend, and then merged with the results returned by each backend. Views and Rules that join across the nodes cannot be created. However, to JOIN tables that access data only in the same node, a VIEW can be made. This VIEW has to be registered in the pgpool_catalog.dist_def table. A col_name and a dist_def_func will have to be registered too. These are used when an insert is performed on the view.

Functions / Triggers (for parallel mode)

The same definition will be created on all the backends for functions. Joining across the nodes, and accessing data on the other nodes cannot be performed inside the functions.

Extended Query Protocol (for parallel mode)

The extended query protocol used by JDBC drivers, etc. is not supported. The simple query protocol must be used. This means you cannot use prepared statements.

Natural Join (for parallel mode)

The Natural Join is not supported. "ON join condition" or "USING (join_column)" must be used.

USING CLAUSE (for parallel mode)

The USING CLAUSE is converted to an ON CLAUSE by the query rewrite process. Therefore, when "*" is used at target list, the joined column(s) appear twice.

Here is an example:

```
=# SELECT * FROM t1 JOIN t2 USING(id);
 id |  t  |   t
----+-----+-------
  1 | 1st | first
(1 row)
```

In the rewrite process "USING" is translated into "ON". So the effective result is as follows:

```
=# SELECT * FROM t1 JOIN t2 ON t1.id = t2.id;
```

```
 id |  t  | id |   t
----+-----+----+-------
  1 | 1st |  1 | first
(1 row)
```

Notice that column "t" is duplicated.

Multi-byte Characters (for all modes)

pgpool-II does not translate between different multi-byte characters. The encoding for the client, backend and System DB must be the same.

Multi-statement Query (for all modes)

pgpool-II cannot process multi-statement queries.

Deadlocks (for parallel mode)

Deadlocks across the backends cannot be detected. For example:

```
(tellers table is partitioned using the following rule)
  tid <= 10  --> node 0
  tid >= 10  --> node 1

A) BEGIN;
B) BEGIN;
A) SELECT * FROM tellers WHERE tid = 11 FOR UPDATE;
B) SELECT * FROM tellers WHERE tid = 1 FOR UPDATE;
A) SELECT * FROM tellers WHERE tid = 1 FOR UPDATE;
B) SELECT * FROM tellers WHERE tid = 11 FOR UPDATE;
```

In the case above, a single node cannot detect the deadlock, so pgpool-II will wait for the response indefinitely. This phenomenon can occur with any query that acquires row level locks.

Also, if a deadlock occurs in one node, transaction states in each node will not be consistent. Therefore, pgpool-II terminates the process if a deadlock is detected.

```
pool_read_kind: kind does not match between master(84) slot[1] (69)
```

Schemas (for parallel mode)

Objects in a schema other than public must be fully qualified like:

```
schema.object
```

pgpool-II cannot resolve the correct schema when the path is set as follows:

```
set search_path = xxx
```

and the schema name is omitted in a query.

table name - column name(for parallel mode)

Limitation object:Parallel mode
A table or a column name cannot start by pool_. When rewriting queries, these names are used by internal processing.

System DB

Partitioning Rules

Only one partitioning key column can be defined in one partitioning rule. Conditions like 'x or y' are not supported.

Environment Requirements

libpq

`libpq` is linked while building pgpool-II. libpq version must be 3.0. Building pgpool-II with libpq version 2.0 will fail.
Also, the System DB must be PostgreSQL 7.4 or later.

Query Cache

With on disk query cache, invalidation must be done manually. This does not apply to on memory query cache. Cache invalidation is automatically done with on memory query cache.

back to top

Internal information

pgpool-II version 2.0.x brings extensive modifications, compared with the version 1.x Please note that what follows doesn't apply to version 1.x.

Parallel execution engine

The parallel execution engine is built into pgpool-II. This engine performs the same Query on each node, and drives the engine that transmits the result to the front end, depending on the nodes' answers.

Query Rewriting

This explains the Query rewriting that pgpool-II does in parallel mode.

In parallel mode, a query transmitted by the client goes through two stages of processing:

- Analysis of Query
- Rewriting of Query

What follows explains these two processing steps:

Analysis of Query

Introduction

The retrieval query submitted by the client goes through the SQL parser. It is then analyzed using information stored in the system DB. Execution status of each part of this query is updated using this information.

This execution status stores where this node can be treated. For instance, if a table's data is distributed on several nodes (as declared in the catalog's pgpool_catalog.dist_def table), it has to be retrieved from all nodes. On the other hand, data from a table registered in pgpool_catalog.replicate_def is replicated, and can therefore be retrieved from any node.

These states are 'P' when data has to be processed by all nodes, 'L' when it should be processed by one node. The 'S' status is a special case: it means that there is another step to be performed on the data after retrieving it from all nodes. For example, sorting data coming from a table registered in the pgpool_catalog.dist_def table.

The retrieval query is analyzed in the following order, and its execution status changes during this processing. Where the query will be processed depends on the final status of the main select.

1. Are UNION, EXTRACT, and INTERSECT used or not?
2. What is the Execution status of FROM clause ?
3. Change the execution status by TARGETLIST
4. Change in execution status according to WHERE clause
5. Change in execution status according to GROUP BY clause
6. Change in execution status according to HAVING clause
7. Change in execution status according to ORDER BY clause
8. Changes into the LIMIT OFFSET predicate in the execution status.
9. Acquisition of the final execution status of SELECT

The relation between the final execution status of SELECT and the processing place is as follows.

Execution status	Processed place
L	Query issued on either node.
P	Returns data to the client by running the same query on all nodes and using the parallel execution engine.
S	After processing using the system DB, data is returned to the client.

The above-mentioned rule also applies to Sub-Query. In the simple following Query, if p1-table is registered in pgpool_catalog.dist_def table of the system DB (p1-table is distributed), the final execution status of the subquery becomes P, and as a result, the parent of the subquery, SELECT, also becomes P.

```
SELECT * FROM (SELECT * FROM P1-table) as P2-table;
```

Next, let's explain how the execution status changes concretely. Let's start with an example, to explain the FROM status.

Execution status of FROM clause

This is a retrieval Query (SELECT). The data set and its status (P,L and S) is defined according to the FROM clause The execution status of the table is as follows: when there is only one table in the from clause, the execution status of the entire dataset is this table's execution status. When there are two or more tables or sub-queries in the FROM clause, the execution is decided according to the join method and the execution statuses, as show in the following table.

JOIN type	LEFT OUTER JOIN			RIGHT OUTER JOIN			FULL OUTER JOIN			Other		
left/right	P	L	S	P	L	S	P	L	S	P	L	S
P	S	P	S	S	S	S	S	S	S	S	P	S
L	S	L	S	P	L	S	S	L	S	P	L	S
S	S	S	S	S	S	S	S	S	S	S	S	S

In the following examples, P1-table is in the P status. L1-table and L2-table are in the L status.

```
SELECT * FROM P1-table,L1-table,L2-table;
```

P1-table (left) and L1-table (right) are joined, and according to the table they get the P status. With this P status, they are joined with the L2-table, in the L status, which is now in the P status too.

Changes in execution status because of TARGETLIST and WHERE clause

In a basic Query, the execution status is the one from the FROM clause. However, if there is a TARGETLIST, the execution status of the WHERE clause can change in the following cases.

1. When there is a subquery
2. When there is an aggregate function or DISTINCT in TARGETLIST marked as 'P'
3. When a column that does not exist in a table (data set) in the FROM clause is used

In these cases, the final execution status of the subquery, the execution status of TARGETLIST and the WHERE clause

get the S status if the initial status was P or S.

In the following example, when the table used by the subquery is P, the final execution status of the subquery gets the P status. Therefore, The execution status of the WHERE clause gets the S status, whatever the execution status of L1 is, and this query is run in the system DB.

```
SELECT * FROM L1-table where L1-table.column IN (SELECT * FROM P1-table);
```

The FROM clause changes to the S status when there is an aggregate function in a 'P' TARGETLIST, in order to perform the aggregate after all data has been acquired. Some optimization on the aggregate function is done under specific conditions.

A column that does not exist in a table can be used in a query. For instance, in the following correlated subquery:

```
SELECT * FROM L1-table WHERE L1-table.col1 IN (SELECT * FROM P1-table WHERE P1-table.col = L1-table.col1);
```

This subquery refers to L1-table.col1, from the L1-table. The execution status of the WHERE clause of the subquery is 'S'.

Change in execution status because of GROUP BY, HAVING, ORDER BY and LIMIT/OFFSET

The execution status of the WHERE clause is changed to 'S' when there is any GROUP BY, HAVING, or ORDER BY clause, or LIMIT/OFFSET predicate and status is 'P'. A query with no GROUP BY clause gets the execution status of the WHERE clause. In the same way, the execution status of the GROUP BY clause is used when there is no HAVING clause. The ORDER BY clause and the LIMIT/OFFSET predicate are also similar in this behavior.

When UNION, EXTRACT, and INTERSECT are used

UNION, EXTRACT, and INTERSECT queries' status depends on the final execution status of both the left and right SELECT statements. If both statements are L, the combined statement is L. If both statements are P, and the query is a UNION ALL the combined statement is P. For any other combination, the resulting status is S.

Acquisition of the final execution status of SELECT

If everything in the SELECT has a status of L, then the final execution status is L. The same rule applies for P. For any other combination, the final status is S. If the status is L, the load is distributed among nodes when loadbalance_mode is true and sent to the MASTER if false. For P, parallel processing is done with the parallel execution engine. For S, the query rewriting presented below is done.

Query rewriting

The Query is rewritten by using the execution status acquired while analyzing the query. Here is an example. The P1-table has the P status, the L1-table has the L status.

```
SELECT P1-table.col, L1-table.col
FROM P1-table,L1-table
where P1-table.col = L1-table.col
order by P1-table.col;
```

In this query, because there is an ORDER BY clause, the status is S. The FROM clause, the WHERE clause, and TARGETLIST are in the P status. The query is rewritten into something like this :

```
SELECT P1-table.col, L1-table.col FROM
    dblink(select pool_parallel(SELECT P1-table.col, L1-table.col FROM P1-table,L1-table where P1-table.col = L1-table.co
      order by P1-table.col;
```

dblink transmits the query to pgpool-II here. the pool_parallel function is responsible for sending the query to the parallel execution engine.

In this example, the FROM and WHERE clause, and the TARGETLIST are run in parallel mode. This isn't the real rewritten query, just something for the sake of providing an example.

Here is another case:

```
SELECT L1-table.col FROM L1-table WHERE L1-table.col % 2 = 0 AND L1-table.col IN (SELECT P1-table FROM P1-table) ;
```

In this example, the FROM and WHERE clause and the TARGETLIST are in the L status. Because the subquery is in the P status, the query itself is in the S status. The rewriting is, as a consequence, performed as follows.

```
SELECT L1-table.col
  FROM dblink(SELECT loadbalance(SELECT L1-table.col
              FROM L1-table
              WHERE L1-table.col % 2 = 0
                AND TRUE))
    WHERE
        L1-table.col %2 = 0 AND
      L1-table.col IN
      (
          SELECT P1-Table FROM
          dblink(select pool_parallel(SELECT P1-table FROM P1-table))
      ) ;
```

pool_loadbalance is a function responsible for dispatching queries to either node.

Query rewriting for aggregate functions

For grouping queries (aggregate functions and GROUP BY), rewriting tries to reduce the load on the system DB by performing a first aggregate on each node.

First, let's see what pgpool-II does for rewriting.

This query has the P status in the FROM clause and count(*) in TARGETLIST. So the rewriting is done as follows.

```
select count(*) from P1-table;

  -> rewrite

  SELECT
      sum(pool_c$1) as count
  FROM
      dblink(select pool_parallel('select count(*) from  P1-table'))
                  AS pool_$1g (pool_c$1 bigint);
```

Query rewriting like above is done in these conditions.

1. The FROM clause is in the P status.
2. The column specified in the aggregate function (only count, sum, min, max, and avg) and GROUP BY is used in the target list.
3. The WHERE clause is in the P status.
4. Only the columns defined by the aggregate function (only count, sum, min, max, and avg), used in the HAVING clause and the FROM clause, and the column specified for GROUP BY are used.

Notes on the parallel mode

The column names and types are needed when a query is analyzed in parallel mode. Therefore, when an expression or a function is used in the TARGETLIST of a subquery, the alias and type (through a cast) are needed. Please note that if no cast is provided in an expression or a function, the text type will be chosen as a default. For count(), bigint type is assumed, and for sum(), numeric. For min()/max(), if the argument is a date type, returned datatype is date, else, it is assumed numeric. avg() is processed as sum()/count() (sum divided by count).

About the performance of a parallel mode

Here is a rough estimate of the query performance versus execution status:

Execution status	Performance
L	There is no performance deterioration with a single node, excluding the overhead of pgpool-II, because there is no parallel querying done at all.
P	Parallel processing is fast, especially the sequential scans. It's easy to get speedups because the scan of a big table becomes the parallel scan of much smaller ones by being distributed on several nodes.
S	When aggregate functions can be rewritten in a parallel fashion, they are fast.

back to top

Tutorial

A tutorial for pgpool-II is available.

back to top