# PoWA Documentation

***Release 4.0.0***

**The PoWA-team**

**Oct 10, 2022**

# Contents

1 Main components 3

**Note:** You can try powa at demo-powa.anayrat.info. Just click "Login" and try its features! Note that in order to get interesting metrics, resources have been limited on this server (2 vCPU, 384MB of RAM and 150iops for the disks). Please be patient when using it.

Thanks to Adrien Nayrat for providing it.

PoWA (PostgreSQL Workload Analyzer) is a performance tool for **PostgreSQL 9.4 and newer** allowing to collect, aggregate and purge statistics on multiple PostgreSQL instances from various *Stats Extensions*.

Depending on your needs, you can either use the provided background worker (requires a PostgreSQL restart, and more suited for single-instance setups), or the provided *PoWA-collector* daemon (does not require a PostgreSQL restart, can gather performance metrics from multiple instances, including standby).

This includes support for various **stat extensions**:

- *pg_stat_statements*, providing data about queries being executed
- *pg_qualstats*, providing data about predicates, or where clauses
- *pg_stat_kcache*, providing data about operating-system level cache
- *pg_wait_sampling*, providing data about wait events

It supports the following extension:

- *HypoPG*, allowing you to create hypothetical indexes and test their usefulness without creating the real index

Additionnaly, the PoWA User Interface allows you to make the most of this information.

CHAPTER 1

# Main components

- **PoWA-archivist** is the PostgreSQL extension, collecting statistics.
- **PoWA-collector** is the daemon that gather performance metrics from remote PostgreSQL instances (optional) on a dedicated repository server.
- **PoWA-web** is the graphical user interface to powa-collected metrics.
- **Stat extensions** are the actual source of data.
- **PoWA** is the whole project.

You should first take a look at the *Quickstart* guide.

## 1.1 Quickstart

**Warning:** The current version of PoWA is designed for PostgreSQL 9.4 and newer. If you want to use PoWA on PostgreSQL < 9.4, please use the 1.x series

**The following describes the installation of the two modules of PoWA:**

- powa-archivist with the PGDG packages (Red Hat/CentOS 6/7, Debian) or from the sources
- powa-web from the PGDG packages (Red Hat/CentOS 7) or with python pip

**Note:** This page shows how to configure a local PoWA setup. If you're interested in configuring PoWA for multiple servers, and/or for standby servers, please also refer to the *Remote setup* page to see the differences in such setups.

### 1.1.1 Install PoWA from packages (Red Hat/CentOS/Debian)

### Prerequisites

PoWA must be installed on the PostgreSQL instance that you are monitoring.

---

**Note:** All extensions except **hypopg** only need to be installed once, in the **powa** database (or another database configured by the configuration option **powa.database**).

hypopg must be installed in every database on which you want to be able to get automatic index suggestion, including the powa database if needed.

powa-web must be configured to connect on the database where you installed all the extensions.

---

We suppose that you are using the packages from the PostgreSQL Development Group (https://yum.postgresql.org/ or https://apt.postgresql.org/). For example for PostgreSQL 9.6 on CentOS 7 a cluster is installed with the following commands:

```
yum install https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-7-x86_64/
↪pgdg-centos96-9.6-3.noarch.rpm
yum install postgresql96 postgresql96-server
/usr/pgsql-9.6/bin/postgresql96-setup initdb
systemctl start postgresql-9.6
```

You will also need the PostgreSQL contrib package to provide the *pg_stat_statements* extension:

```
yum install postgresql96-contrib
```

On Debian, that would be:

```
apt-get install postgresql-9.6 postgresql-client-9.6 postgresql-contrib-9.6
```

In these examples and the following ones, replace 9.6 or 96 according to your version (11, 10, 9.5…).

### Installation of the PostgreSQL extensions

You can simply install the packages provided by the PGDG repository according to your PostgreSQL version. For example on Red Hat/CentOS for PostgreSQL 9.6:

```
yum install powa_96 pg_qualstats96 pg_stat_kcache96 hypopg_96
```

On Debian, this will be:

```
apt-get install postgresql-9.6-powa postgresql-9.6-pg-qualstats postgresql-9.6-pg-
↪stat-kcache postgresql-9.6-hypopg
```

On other systems, or to test newer unpackaged version, you will have to compile some extensions manually *as described in the next section*:

```
apt-get install postgresql-9.6-powa
```

Once all extensions are installed or compiled, add the required modules to *shared_preload_libraries* in the *postgresql.conf* of your instance:

```
shared_preload_libraries='pg_stat_statements,powa,pg_stat_kcache,pg_qualstats'
```

**Note:** If you also installed the pg_wait_sampling extension, don't forget to add it to `shared_preload_libraries` too.

Now restart PostgreSQL. Under RHEL/CentOS 6 (as root):

```
/etc/init.d/postgresql-9.6 restart
```

Under RHEL/CentOS 7:

```
systemctl restart postgresql-9.6
```

On Debian:

```
pg_ctlcluster 9.6 main restart
```

Log in to your PostgreSQL as a superuser and create a *powa* database:

```
CREATE DATABASE powa ;
```

Create the required extensions in this new database:

```
\c powa
CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION btree_gist;
CREATE EXTENSION powa;
CREATE EXTENSION pg_qualstats;
CREATE EXTENSION pg_stat_kcache;
```

**Note:** If you also installed the pg_wait_sampling extension, don't forget to create the extension too.

PoWA needs the *hypopg* extension in all databases of the cluster in order to check that the suggested indexes are efficient:

```
CREATE EXTENSION hypopg;
```

One last step is to create a role that has superuser privileges and is able to login to the cluster (use your own credentials):

```
CREATE ROLE powa SUPERUSER LOGIN PASSWORD 'astrongpassword' ;
```

The Web UI requires you to log in with a PostgreSQL role that has superuser privileges as only a superuser can access to the query text in PostgreSQL. PoWA follows the same principle.

PoWA is now up and running on the PostgreSQL-side. You still need to set up the web interface in order to access your history. By default powa-archivist stores history for 1 day and takes a snapshot every 5 minutes. These default settings can be easily changed afterwards.

### Install the Web UI

The RPM packages work for now only on Red Hat/CentOS 7. For Red Hat/CentOS 6 or Debian, see *the installation through pip* or *the full manual installation guide*.

You can install the web client on any server you like. The only requirement is that the web client can connect to the previously set up PostgreSQL cluster.

If you're setting up PoWA on another server, you have to install the PGDG repo package again. This is required to install the *powa_96-web* package and some dependencies.

Again, for example for PostgreSQL 9.6 on CentOS 7:

```
yum install https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-7-x86_64/
↪pgdg-centos96-9.6-3.noarch.rpm
```

Install the *powa_96-web* RPM package with its dependencies:

```
yum install powa_96-web
```

Create the */etc/powa-web.conf* config-file to tell the UI how to connect to your freshly installed PoWA database. Of course, change the given cookie to something from your own. For example to connect to the local instance on *localhost*:

```
servers={
  'main': {
    'host': 'localhost',
    'port': '5432',
    'database': 'powa'
  }
}
cookie_secret="SUPERSECRET_THAT_YOU_SHOULD_CHANGE"
```

Don't forget to allow the web server to connect to the PostgreSQL cluster, and edit your *pg_hba.conf* accordingly.

Then, run powa-web:

```
powa-web
```

The Web UI is now available on port 8888, for example on http://localhost:8888/. You may have to configure your firewall to open the access to the outside. Use the role created earlier in PostgreSQL to connect to the UI.

### 1.1.2 Build and install powa-archivist from the sources

#### Prerequisites

You will need a compiler, the appropriate PostgreSQL development packages, and some contrib modules.

While on most installation, the contrib modules are installed with a postgresql-contrib package, if you wish to install them from source, you should note that only the following modules are required:

- btree_gist
- pg_stat_statements

On Red Hat/CentOS:

```
yum install postgresql96-devel postgresql96-contrib
```

On Debian:

```
apt-get install postgresql-server-dev-9.6 postgresql-contrib-9.6
```

### Installation

Download powa-archivist latest release:

```
wget https://github.com/powa-team/powa-archivist/archive/REL_4_0_0.tar.gz
```

Convenience scripts are offered to build every project that PoWA can take advantage of.

First, the install_all.sql file:

```
CREATE DATABASE IF NOT EXISTS powa;
\c powa
CREATE EXTENSION IF NOT EXISTS btree_gist;
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
CREATE EXTENSION IF NOT EXISTS pg_stat_kcache;
CREATE EXTENSION IF NOT EXISTS pg_qualstats;
CREATE EXTENSION IF NOT EXISTS pg_wait_sampling;
CREATE EXTENSION IF NOT EXISTS powa;
```

And the main build script:

```
#!/bin/bash
# This script is meant to install every PostgreSQL extension compatible with
# PoWA.
wget https://github.com/powa-team/pg_qualstats/archive/1.0.7.tar.gz -O pg_
↪qualstats-1.0.7.tar.gz
tar zxvf pg_qualstats-1.0.7.tar.gz
cd pg_qualstats-1.0.7
(make && sudo make install)  > /dev/null 2>&1
cd ..
rm pg_qualstats-1.0.7.tar.gz
rm pg_qualstats-1.0.7 -rf
wget https://github.com/powa-team/pg_stat_kcache/archive/REL2_1_1.tar.gz -O␣
↪pg_stat_kcache-REL2_1_1.tar.gz
tar zxvf pg_stat_kcache-REL2_1_1.tar.gz
cd pg_stat_kcache-REL2_1_1
(make && sudo make install)  > /dev/null 2>&1
cd ..
rm pg_stat_kcache-REL2_1_1.tar.gz
rm pg_stat_kcache-REL2_1_1 -rf
(make && sudo make install)  > /dev/null 2>&1
cd ..
wget https://github.com/postgrespro/pg_wait_sampling/archive/v1.1.tar.gz -O␣
↪pg_wait_sampling-v1.1.tar.gz
tar zxvf pg_wait_sampling-v1.1.tar.gz
cd pg_wait_sampling-v1.1
(make && sudo make install)  > /dev/null 2>&1
cd ..
rm pg_wait_sampling-v1.1.tar.gz
rm pg_wait_sampling-v1.1 -rf
echo ""
echo "You should add the following line to your postgresql.conf:"
echo ''
echo "shared_preload_libraries='pg_stat_statements,powa,pg_stat_kcache,pg_
↪qualstats,pg_wait_sampling'"
echo ""
```

```
echo "Once done, restart your postgresql server and run the install_all.sql␣
↪file"
echo "with a superuser, for example: "
echo "  psql -U postgres -f install_all.sql"
```

This script will ask for your super user password, provided the sudo command is available, and install powa, pg_qualstats, pg_stat_kcache and pg_wait_sampling for you.

**Warning:** This script is not intended to be run on a production server, as it compiles all the extensions. You should prefer to install packages on your production servers.

Once done, you should modify your PostgreSQL configuration as mentioned by the script, putting the following line in your *postgresql.conf* file:

```
shared_preload_libraries='pg_stat_statements,powa,pg_stat_kcache,pg_qualstats,pg_wait_
↪sampling'
```

Optionally, you can install the hypopg extension the same way from https://github.com/hypopg/hypopg/releases.

And restart your server, according to your distribution's preferred way of doing so, for example:

Init scripts:

```
/etc/init.d/postgresql-9.6 restart
```

Debian pg_ctlcluster wrapper:

```
pg_ctlcluster 9.6 main restart
```

Systemd:

```
systemctl restart postgresql
```

The last step is to create a database dedicated to the PoWA repository, and create every extension in it. The install_all.sql file performs this task:

```
psql -U postgres -f install_all.sql
CREATE DATABASE
You are now connected to database "powa" as user "postgres".
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
```

### 1.1.3 Install powa-web anywhere

You do not have to install the GUI on the same machine your instance is running.

#### Prerequisites

- The Python language, either 2.6, 2.7 or > 3
- The Python language headers, either 2.6, 2.7 or > 3

- The pip installer for Python. It is usually packaged as "python-pip", for example:

Debian:

```
sudo apt-get install python-pip python-dev
```

Red Hat/CentOS:

```
sudo yum install python-pip python-devel
```

### Installation

To install powa-web, just issue the following comamnd:

```
sudo pip install powa-web
```

Then you'll have to configure a config file somewhere, in one of those location:

- /etc/powa-web.conf
- ~/.config/powa-web.conf
- ~/.powa-web.conf
- ./powa-web.conf

The configuration file is a simple JSON one. Copy the following content to one of the above locations and modify it according to your setup:

```
servers={
  'main': {
    'host': 'localhost',
    'port': '5432',
    'database': 'powa'
  }
}
cookie_secret="SUPERSECRET_THAT_YOU_SHOULD_CHANGE"
```

The servers key define a list of server available for connection by PoWA-web. You should ensure that the pg_hba.conf file is properly configured.

The cookie_secret is used as a key to crypt cookies between the client and the server. You should DEFINITELY not keep the default if you value your security.

Other options are described in *the full manual installation guide*.

Then, run powa-web:

```
powa-web
```

The UI is now available on the 8888 port (eg. http://localhost:8888). Login with the credentials of the *powa* PostgreSQL user.

## 1.2 Remote setup

Before **version 4**, all the performance data collected were stored locally. This had two majors drawbacks:

- it adds a non negligeable performance cost, both when collecting data and when using the user interface

- it's not possible to collect data on hot-standby servers

With version 4, it's now possible to store the data of one or multiples servers on an external PostgreSQL database. This chapter describes how to configure such remote mode.

### 1.2.1 What did not change

Only the storage part changed. Therefore, it's still mandatory to configure at least *pg_stat_statements* on each PostgreSQL instance, and all the other *Stats Extensions* you want to use. The list of extension can of course be different on each instance.

### 1.2.2 Setup the main repository database

A PostgreSQL 9.4 or upward is required. Ideally, you should setup a dedicated instance for storing the PoWA performance data, especially if you want to setup more than a few remote servers.

You need to setup a dedicated database and install the latest version of *PoWA archivist*. The *Installation* and *background worker configuration* documentation will explain in detail how to do so.

However, please note that if you don't want to gather performance data for the repository PostgreSQL server, the *shared_preload_libraries* configuration and instance restart is not required anymore.

### 1.2.3 Configure PoWA and stats extensions on each remote server

You need to configure *PoWA archivist* and the *Stats Extensions* of your choice on each remote PostgreSQL server.

### 1.2.4 Declare the list of remote servers and their extensions

*PoWA archivist* provides some SQL functions for that.

You most likely want to declare a *remote sever* using the *powa_register_server* function. For instance:

```
SELECT powa_register_server(hostname => 'myserver.domain.com',
    alias => 'myserver',
    password => 'mypassword',
    extensions => '{pg_stat_kcache,pg_qualstats,pg_wait_sampling}');
```

You can consult the *Remote servers configuration* page for a full documentation of the available SQL API.

### 1.2.5 Configure powa-collector

Do all the required configuration as documented in *PoWA-collector*.

Then you can check that everything is working by simply launching the collector. For instance:

```
./powa-collector.py
```

**Warning:** It's highly recommended to configure powa-collector as a daemon, with any facility provided by your operating system, once the initial setup and testing is finished.

Gathering of remote data will start, as described by previous configuration.

### 1.2.6 Configure the User Interface

You can follow the *PoWA-web* documentation. Obviously, in case of remote setup you only need to configure a single connection information per PoWA remote repository.

Once all those steps are finished, you should have a working remote setup for PoWA!

## 1.3 Frequently Asked question

### 1.3.1 Can I use PoWA on a standby server, or store the data on an external server

Yes! Since version 4 of PoWA, it's possible to setup a **remote snapshot**, thus aggregating all the performance data on a dedicated remote PostgreSQL server. This mode greatly limits the performance impact of PoWA on the configured servers, and also allows to use PoWA on standby servers too. See the *Remote setup* documentation more details.

### 1.3.2 Some queries don't show up in the UI

That's a know limitation with the current implementation of powa-web.

For now, the UI will only display information about queries that have been run on **at least** two distinct snapshots of powa-archivist (parameter *powa.frequency*). With default settings, that means you need to run activity for at least 10 minutes.

This is however usually not a problem since queries only executed a few time and never again are not really a target for optimization.

### 1.3.3 I ran some queries and index suggestion doesn't suggest any index

With default configuration, pg_qualstats will only sample 1% of the queries. This default value is a safeguard to avoid overhead on heavily loaded production server. However, if you're just doing some test that means that you'll miss most of the WHERE and JOIN clauses, and index suggestion won't be able to suggest indexes.

If you want pg_qualstats to sample every query, you need to configure *pg_qualstats.sample_rate = 1* in the **postgresql.conf** configuration file, and reload the configuration.

Please keep in mind that such a configuration can have a strong impact on the performance, especially if a lot of concurrent and fast queries are executed.

## 1.4 Security

**Warning: You need to be careful about the security of your PostgreSQL instance when installing PoWA.**

We designed POWA so that the user interface will only communicate with PostgreSQL via prepared statements. This will prevent the risk of SQL injection.

However to connect to the PoWA User Interface, you will use the login and password of a PostgreSQL user. If you don't protect your communications, an attacker placed between the GUI and PostgreSQL, or between you and the GUI, could gain your user rights to your database server.

Therefore we **strongly** recommend the following precautions:

- Read the Great PostgreSQL Documentation
- Check your *pg_hba.conf* file
- Do not allow users to access PoWA from the Internet
- Do not allow users to access PostgreSQL from the Internet
- Run PoWA on a HTTPS server and disable HTTP access
- Use SSL to protect the connection between the GUI and PostgreSQL
- Reject unprotected connections between the GUI and PostgreSQL (*hostnossl .... reject*)
- Check your *pg_hba.conf* file again

Please also note that you need to manually authorize the roles to see the data in the powa database. For instance, you might run:

```
powa=# GRANT SELECT ON ALL TABLES IN SCHEMA public TO ui_user;
powa=# GRANT SELECT ON pg_statistic TO ui_user;
```

### 1.4.1 User objects

powa-web will connect to the databases you select to help you optimize them.

Therefore, for each postgres roles using powa, you also need to:

- grant **SELECT** privilege on the pg_statistic and the user tables (don't forget tables that aren't in the public schema).
- give **CONNECT** privilege on the databases.

If you don't, some useful parts of the UI won't work as intended.

### 1.4.2 Connection on remote servers

With PoWA version 4 and newer, you can register *remote servers* in the *powa_servers* table (usually using the *powa_register_server* function).

This table can optionally store a **password** to connect on this remote server. If the password is NULL, the connection will then be attempted using the authentication method that libpq supports of your choice.

Storing a plain text password in this table is definitely **NOT** a best practice, and we encourage you to rely on the other libpq authentication methods.

## 1.5 Components

This sections gathers the various components and external extensions that PoWA can use.

### 1.5.1 PoWA archivist

#### Installation

### Prerequisites

- PostgreSQL >= 9.4
- PostgreSQL contrib modules (pg_stat_statements and btree_gist)
- PostgreSQL server headers

On Debian, the PostgreSQL server headers are installed via the `postgresql-server-dev-X.Y` package:

```
apt-get install postgresql-server-dev-9.4 postgresql-contrib-9.4
```

On RPM-based distros:

```
yum install postgresql94-devel postgresql94-contrib
```

You also need a C compiler and other standard development tools.

On Debian, these can be installed via the `build-essential` package:

```
apt-get install build-essential
```

On RPM-based distros, the "Development Tools" can be used:

```
yum groupinstall "Development Tools"
```

### Installation

Grab the latest release, and install it:

```
wget https://github.com/powa-team/powa-archivist/archive/REL_4_0_0.tar.gz -O␣
↪powa-archivist-REL_4_0_0.tar.gz
tar zxvf powa-archivist-REL_4_0_0.tar.gz
cd powa-archivist-REL_4_0_0
```

Compile and install it:

```
make
sudo make install
```

It should output something like the following :

```
/bin/mkdir -p '/usr/share/postgresql-9.4/extension'
/bin/mkdir -p '/usr/share/postgresql-9.4/extension'
/bin/mkdir -p '/usr/lib64/postgresql-9.4/lib64'
/bin/mkdir -p '/usr/share/doc/postgresql-9.4/extension'
/usr/bin/install -c -m 644 powa.control '/usr/share/postgresql-9.4/extension/'
/usr/bin/install -c -m 644 powa--2.0.sql '/usr/share/postgresql-9.4/extension/'
/usr/bin/install -c -m 644 README.md '/usr/share/doc/postgresql-9.4/extension/'
/usr/bin/install -c -m 755  powa.so '/usr/lib64/postgresql-9.4/lib64/'
```

Create the PoWA database and create the required extensions, with the following statements:

```
CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION btree_gist;
CREATE EXTENSION powa;
```

Example:

```
bash-4.1$ psql
psql (9.3.5)
Type "help" for help.
postgres=# create database powa;
CREATE DATABASE
postgres=# \c powa
You are now connected to database "powa" as user "postgres".
powa=# create extension pg_stat_statements ;
CREATE EXTENSION
powa=# create extension btree_gist ;
CREATE EXTENSION
powa=# create extension powa;
CREATE EXTENSION
```

As PoWA-archivist is implemented as a background worker, the library must be loaded at server start time.

For this, modify the `postgresql.conf` configuration file, and add powa and pg_stat_statements to the `shared_preload_libraries` parameter:

```
shared_preload_libraries = 'pg_stat_statements,powa'
```

If possible, activate `track_io_timing` too:

```
track_io_timing = on
```

PostgreSQL should then be restarted.

**Warning:** Since PoWA 4, you need to specify **powa** in the *shared_preload_libraries* configuration **ONLY** if you want to store the performance data locally. For remote storage, please see the *Remote setup* documentation. The *pg_stat_statements* extension (as all other *Stats Extensions*) still required to be configured in the *shared_preload_libraries* setting.

If you're setting up a repository database for a remote server, you can also entirely skip the *pg_stat_statements* configuration and the restart.

### background worker configuration

**Note:** This is intended for local-mode setup.

The following configuration parameters (GUCs) are available in `postgresql.conf`:

**powa.frequency:** Defaults to `5min`. Defines the frequency of the snapshots, in milliseconds or any time unit supported by PostgreSQL. Minimum 5s. You can use the usual postgresql time abbreviations. If not specified, the unit is seconds. Setting it to -1 will disable powa (powa will still start, but it won't collect anything anymore, and wont connect to the database).

**powa.retention:** Defaults to `1d` (1 day) Automatically purge data older than that. If not specified, the unit is minutes.

**powa.database:** Defaults to `powa` Defines the database of the workload repository.

**powa.coalesce:** Defaults to `100`. Defines the amount of records to group together in the table.

### Remote servers configuration

**Note:** This is intended for the *Remote setup* mode.

You can declare, configure and remove *remote servers* using an SQL API.

#### powa_register_server

This function declare a new remote server and the activated extensions.

The arguments are:

**hostname (*text*):** Mandatory, default *NULL*. Hostname or IP address of the remote PostgreSQL instance.

**port (*integer*)** Mandatory, default *5432*. Port of the remote PostgreSQL instance.

**alias (*text*):** Optional, default *NULL*. User-friendly alias of the remote PostgreSQL instance (needs to be unique).

**username (*text*):** Mandatory, default *'powa'*. Username to user to connect on the remote PostgreSQL instance.

**password (*text*):** Optional, default *NULL*. Password to user to connect on the remote PostgreSQL instance. If no password is provided, the connection can fallback on other standard authentication method (.pgpass file, certificate...) depending on how the remote server is configured.

**dbname (*text*):** Mandatory, default *'powa'*. Database to connect on the remote PostgreSQL instance.

**frequency (*integer*):** Mandatory, default *300*, Snapshot interval for the remote server, in seconds.

**retention (*interval*):** Mandatory, default *'1 day'::interval*. Data retention for the remote server.

**extensions (*text[]*):** Optional, default *NULL*. List of extensions on the remote server for which the data should be stored. You don't need to specify *pg_stat_statements*. As it's a mandatory extensions, it'll be automatically added.

This function return **true** if the server was registered.

**Note:**

- The (hostname, port) must be unique.
- This function will not try to connect on the remote server to validate that the list of extensions is correct. If you declared extensions that are not available or properly setup on the remote server, the underlying data won't be available and you'll see errors in the *PoWA-collector* logs and the *PoWA-web* user interface.

**Warning:** Connection on the remote server can be attempted by the *PoWA-web* user interface and *PoWA-collector*. The connection for *PoWA-collector* **is mandatory**. The user interface can work without such remote connection, but with **limited features** (notably, index suggestion will not be available).

You can call this function as any SQL function, using a **superuser**.

For instance, to add a remote server on **myserver.domain.com**, with the alias **myserver**, with default port and database, the password **mypassword**, and **all the supported extensions**:

Example:

```
SELECT powa_register_server(hostname => 'myserver.domain.com',
    alias => 'myserver',
    password => 'mypassword',
    extensions => '{pg_stat_kcache,pg_qualstats,pg_wait_sampling}');
```

### powa_activate_extension

This function is automatically called by *powa_register_server*. It can be useful if you setup an additional *Stats Extensions* after the inital *remote server* declaration.

The arguments are:

**_srvid (*integer*):** Mandatory, default *NULL*. Interval serveur identifier. You can find the identifier in the *powa_servers* table, containing the list of remote instances.

**_extname (*text*):** Mandatory, default *NULL*. The name of the extension to activate.

This function return **true** if the extension was activated on the given *remote server*.

Example:

```
SELECT powa_activate_extension(1, 'extension_name');
```

### powa_deactivate_extension

This function can be useful if you removed a *Stats Extensions* after the inital *remote server* declaration.

The arguments are:

**_srvid (*integer*):** Mandatory, default *NULL*. Interval serveur identifier. You can find the identifier in the *powa_servers* table, containing the list of remote instances.

**_extname (*text*):** Mandatory, default *NULL*. The name of the extension to deactivate.

This function return **true** if the extension was deactivated on the given *remote server*.

Example:

```
SELECT powa_deactivate_extension(1, 'extension_name');
```

### powa_configure_server

This function can be useful if you want to change any of the *remote server* property after its inital declaration.

The arguments are:

**_srvid (*integer*):** Mandatory, default *NULL*. Interval serveur identifier. You can find the identifier in the *powa_servers* table, containing the list of remote instances.

**_data (*json*):** Mandatory The changes you want to perform, provided as a JSON value where the key is the property to update and the value is the value to use.

This function return **true** if the configuration was changed for the given *remote server*.

Example:

```
SELECT powa_configure_server(1, '{"alias": "my new alias", "password": null}');
```

### powa_deactivate_server

This function can be useful if you want to disable snapshots on the specified *remote server*, but keep its stored data.

The arguments are:

**_srvid (*integer*):** Mandatory, default *NULL*. Interval serveur identifier. You can find the identifier in the *powa_servers* table, containing the list of remote instances.

This function return **true** if the given *remote server* were deactivated.

Example:

```
SELECT powa_deactivate_server(1);
```

### powa_delete_and_purge_server

This function can be useful if you want to delete a server from the list of *remote servers*, and delete any stored data related to it.

The arguments are:

**_srvid (*integer*):** Mandatory, default *NULL*. Interval serveur identifier. You can find the identifier in the *powa_servers* table, containing the list of remote instances.

This function return **true** if the given *remote server* were deleted.

Example:

```
SELECT powa_delete_and_purge_server(1);
```

### Integrating another stat extension in Powa

Clone the repository:

```
git clone https://github.com/powa-team/powa-archivist/
cd powa-archivist/
make && sudo make install
```

Any modification to the background-worker code will need a PostgreSQL restart.

In order to contribute another source of data, you will have to implement the following functions:

**snapshot:** This function is responsible for taking a snapshot of the data source data, and store it somewhere. Usually, this is done in a staging table named **powa_my_data_source_history_current**. It will be called every *powa.frequency* seconds. The function signature looks like this:

```
CREATE OR REPLACE FUNCTION powa_my_data_source_snapshot() RETURNS void AS $PROC$
...
$PROC$ language plpgsql;
```

**aggregate:** This function will be called after every *powa.coalesce* number of snapshots. It is responsible for aggregating the current staging values into another table, to reduce the disk usage for PoWA. Usually, this will be done in an aggregation table named **powa_my_data_source_history**. The function signature looks like this:

```
CREATE OR REPLACE FUNCTION powa_my_data_source_aggregate() RETURNS void AS $PROC$
...
$PROC$ language plpgsql;
```

**purge:** This function will be called after every 10 aggregates and is responsible for purging stale data that should not be kept. The function should take the *powa.retention* global parameter into account to prevent removing data that would still be valid.

```
CREATE OR REPLACE FUNCTION powa_my_data_source_aggregate() RETURNS void AS $PROC$
...
$PROC$ language plpgsql;
```

**unregister:** This function will be called if the related extension is dropped.

Please note that the **module** name used in the **powa_functions** table has to be the same as the extension name, otherwise the function will not be called.

This function should at least remove entries from **powa_functions** table. A minimal function would look like this:

```
CREATE OR REPLACE function public.powa_my_data_source_unregister() RETURNS bool AS
$_$
BEGIN
    DELETE FROM public.powa_functions WHERE module = 'my_data_source';
        RETURN true;
END;
$_$
language plpgsql;
```

Each of these functions should then be registered:

```
INSERT INTO powa_functions (module, operation, function_name, added_manually)
VALUES  ('my_data_source', 'snapshot',   'powa_mydatasource_snapshot',   true),
        ('my_data_source', 'aggregate',  'powa_mydatasource_aggregate',  true),
        ('my_data_source', 'unregister', 'powa_mydatasource_unregister', true),
        ('my_data_source', 'purge',      'powa_mydatasource_purge',      true);
```

### 1.5.2 PoWA-collector

#### Installation

You can install PoWA-collector either using pip or manually.

On Centos 6, you can avoid installing the header files for Python and PostgreSQL by using the package for psycopg2:

```
yum install python-pip python-psycopg2
pip install powa-collector
```

#### Manual install

You'll need the following dependencies:

- python 2.6, 2.7 or > 3
- psycopg2

**debian**

```
apt-get install python python-psycopg2
```

**archlinux**

```
pacman -S python python-psycopg2
```

**fedora**

```
TODO
```

Then, download the latest release on pypi, uncompress it, and copy the sample configuration file:

```
wget https://pypi.io/packages/source/p/powa-collector/powa-collector-0.0.1.
tar.gz
tar -zxvf powa-collector-0.0.1.tar.gz
cd powa-collector-0.0.1
cp ./powa-collector.conf-dist ./powa-collector.conf
./powa-collector
```

Then, jump on the next section to configure powa-collector.

### Configuration

The powa-collector configuration is stored as a simple JSON file. Powa-collector will search its config as either of these files, in this order:

- /etc/powa-collector.conf
- ~/.config/powa-collector.conf
- ~/.powa-collector.conf
- ./powa-collector.conf

The following options are required:

**repository.dsn (string):** An URI to tell powa-collector how to connect on the dedicated repository powa database where to store data for all remote instances.

The following options are optional:

**debug (boolean):** A boolean to specify whether powa-collector should be launched in debug mode, providing a more verbose output, useful for debug purpose.

Example configuration file:

```
{
    "repository": {
        "dsn": "postgresql://powa_user@localhost:5432/powa"
    },
    "debug": false
}
```

**Warning:** The collector needs to be able to connect on the **repository server** and all the declared **remote servers**.

#### Usage

To start the program, simply run the powa-collector.py program. A `SIGTERM` or a `Keyboard Interrupt` on the program will cleanly stop all the thread and exit the program. A `SIGHUP` will reload the configuration.

See also:

#### Protocol

A minimal communication protocol is implented, using the LISTEN/NOTIFY facility provided by postgres, which is used by the powa-web project. You can send queries to collector by sending messages on the "powa_collector" channel. The collector will send answers on the channel you specified, so make sure to listen on it before sending any query to not miss answers.

The requests are of the following form:

COMMAND RESPONSE_CHANNEL OPTIONAL_ARGUMENTS

- COMMAND: mandatory argument describing the query. The following commands are supported:
  - RELOAD: reload the configuration and report that the main thread successfully received the command. The reload will be attempted even if no response channel was provided.
  - WORKERS_STATUS: return a JSON (srvid is the key, status is the content) describing the status of each remote server thread. Command is ignored if no response channel was provided. This command accept an optional argument to get the status of a single remote server, identified by its srvid. If no worker exists for this server, an empty JSON will be returned.
- RESPONSE_CHANNEL: mandatory argument to describe the NOTIFY channel the client listens a response on. '-' can be used if no answer should be sent.
- OPTIONAL_ARGUMENTS: space separated list of arguments, specific to the underlying command.

The answers are of the form:

COMMAND STATUS DATA

- COMMAND: same as the command in the query
- STATUS: OK or KO.
- DATA: reason for the failure if status is KO, otherwise the data for the answer.

### 1.5.3 PoWA-web

#### Installation

You can install PoWA-web either using pip or manually.

On Centos 6, you can avoid installing the header files for Python and PostgreSQL by using the package for psycopg2:

```
yum install python-pip python-psycopg2
pip install powa-web
```

### Manual install

You'll need the following dependencies:

- python 2.6, 2.7 or > 3
- psycopg2
- sqlalchemy >= 0.8.0
- tornado >= 2.0

**debian**

```
apt-get install python python-psycopg2 python-sqlalchemy python-tornado
```

**archlinux**

```
pacman -S python python-psycopg2 python-sqlalchemy python-tornado
```

**fedora**

```
TODO
```

Then, download the latest release on pypi, uncompress it, and copy the sample configuration file:

```
wget https://pypi.io/packages/source/p/powa-web/powa-web-4.0.0.tar.gz
tar -zxvf powa-web-4.0.0.tar.gz
cd powa-web-4.0.0
cp ./powa-web.conf-dist ./powa-web.conf
./powa-web
```

Then, jump on the next section to configure powa-web.

**Note:** If you need to install *powa-web* on CentOS 6, here's a workaround to install sqlalchemy 0.8:

- An RPM can be found at this address
- After installing the RPM, it's required to perform

```
ln -s /usr/lib64/python2.6/site-packages/SQLAlchemy-0.8.2-py2.6-linux-x86_64.egg/
↪sqlalchemy /usr/lib64/python2.6/site-packages/
```

### Configuration

The powa-web configuration is stored as a simple python file. Powa-web will search its config as either of these files, in this order:

- /etc/powa-web.conf
- ~/.config/powa-web.conf

- ~/.powa-web.conf
- ./powa-web.conf

You'll then be noticed of the address and port on which the UI is available. The default is 0.0.0.0:8888, as indicated in this message:

- [I 161105 20:27:39 powa-web:12] Starting powa-web on 0.0.0.0:8888

The following options are required:

**servers (dict):** A dictionary mapping server names to connection information.

```
servers={
  'main': {
    'host': 'localhost',
    'port': '5432',
    'database': 'powa'
  }
}
```

**Warning:**

If any of your databases is not in **utf8** encoding, you should specify a client_encoding option as shown below. This requires at least psycopg2 version 2.4.3

```
servers={
  'main': {
    'host': 'localhost',
    'port': '5432',
    'database': 'powa',
    'query': {'client_encoding': 'utf8'}
  }
}
```

**Note:**

You can set a username and password to allow logging into powa-web without providing credentials. In this case, the powa-web.conf file must be modified like this:

```
servers={
  'main': {
    'host': 'localhost',
    'port': '5432',
    'database': 'powa',
    'username' : 'pg_username',
    'password' : 'the password',
    'query': {'client_encoding': 'utf8'}
  }
}
```

**cookie_secret (str):** A secret key used to secure cookies transiting between the web browser and the server.

```
cookie_secret="SECRET_STRING"
```

The following options are optional:

**port (int):** The port on which the UI will be available (default 8888)

**address (str):** The IP address on which the UI will be available (default 0.0.0.0)

See also:

### Deployment Options

### Apache

PoWA can easily be deployed using Apache mod_wsgi module.

First you have to install and configure Powa like in the *quickstart* section. Check that the powa-web executable works before proceeding.

In your apache configuration file, you should:

- load the mod_wsgi module
- configure it.

The various python3.4 version in the paths below should be set your actual python version:

```
LoadModule wsgi_module modules/mod_wsgi.so
<VirtualHost *:80>
  ServerName myserver.example.com

  DocumentRoot /var/www/

  ErrorLog /var/log/httpd/powa.error.log
  CustomLog /var/log/httpd/powa.access.log combined

  WSGIScriptAlias / /usr/lib/python3.4/site-packages/powa/powa.wsgi

  Alias /static /usr/lib/python3.4/site-packages/powa/static/
</VirtualHost>
```

### Development

This page acts as a central hub for resources useful for PoWA developers.

### PoWA-Web

This section only covers the most simple changes one would want to make to PoWA. For more comprehensive documentation, see the Powa-Web project documentation itself.

Clone the repository:

```
git clone https://github.com/powa-team/powa-web/
cd powa/
make && sudo make install
```

To run the application, use run_powa.py, which will run powa in debug mode. That means the javascript files will not be minified, and will not be compiled into one giant source file.

CSS files are generated using *sass <http://sass-lang.com>*. Javascript files are splitted into AMD modules, which are managed by *requirejs <http://requirejs.org/>* and compiled using *grunt <http://gruntjs.com>*.

These projects depend on NodeJS, and NPM, its package manager, so make sure you are able to install them on your distribution.

Install the development dependencies:

```
npm install -g grunt-cli
npm install .
```

Then, you can run `grunt` to update only the css files, or regenerate optimized javascript builds with `grunt dist`.

### 1.5.4 Stats Extensions

The PoWA-archivist collects data from various stats extensions. To be used in PoWA, a stat extensions has to expose a number of PL/pgSQL functions as stated in *Integrating another stat extension in Powa*.

Currently, the list of supported stat extensions is as follows:

#### pg_stat_statements

The pg_stat_statements extension records statistics of all SQL queries (aka "statements") executed on a given PostgreSQL server.

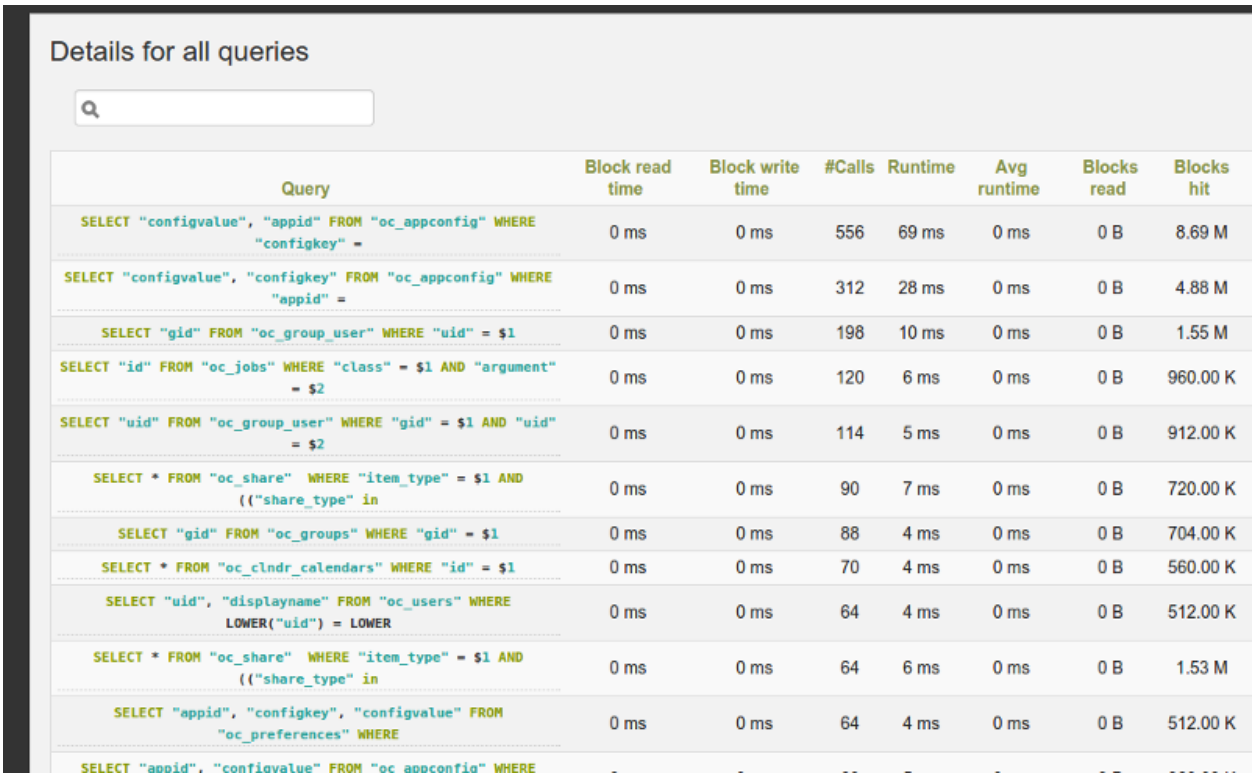
The statistics gathered are available in view called `pg_stat_statements`. This view contains one row for each distinct database ID, user ID and query ID. However the number of distinct statements tracked cannot exceed a certain limit (5 000 by default)

The pg_stat_statements extension is a key component of the PoWA Suite, installing it is **mandatory**.

#### Where is it used in powa-web ?

The PoWA user interface (`powa-web`) relies heavily on pg_stat_statements, so you'll see it used in almost every screen of the tool.

The most useful feature is probably the "Query details" chart which show advanced statistics for each SQL query.




### Installation

pg_stat_statements is an official extension and it is released along with other extensions in the official PostgreSQL packages. You will find it in the `contrib` folder. Depending on which Operating System, you're using you may need to install a separate package to use it. For instance, on `debian` you may need to install the `postgresql-contrib` package.

Then you just have to declare the extension in the `postgresql.conf` file, like this :

```
shared_preload_libraries = 'pg_stat_statements'
```

Restart the PostgreSQL server to reload the libraries.

Connect to the server as a superuser and type:

```
CREATE EXTENSION pg_stat_statements
```

### Configuration

There's a few parameters that you can add to the `postgresql.conf`. For instance you can increase the track limit and allow PostgreSQL to record 10 000 distinct queries:

```
pg_stat_statements.max = 10000
```

For more information about the pg_stat_statements, please read the PostgreSQL documentation:

http://www.postgresql.org/docs/current/static/pgstatstatements.html

### Examples

### See Also

- http://www.craigkerstiens.com/2013/01/10/more-on-postgres-performance/

### pg_qualstats

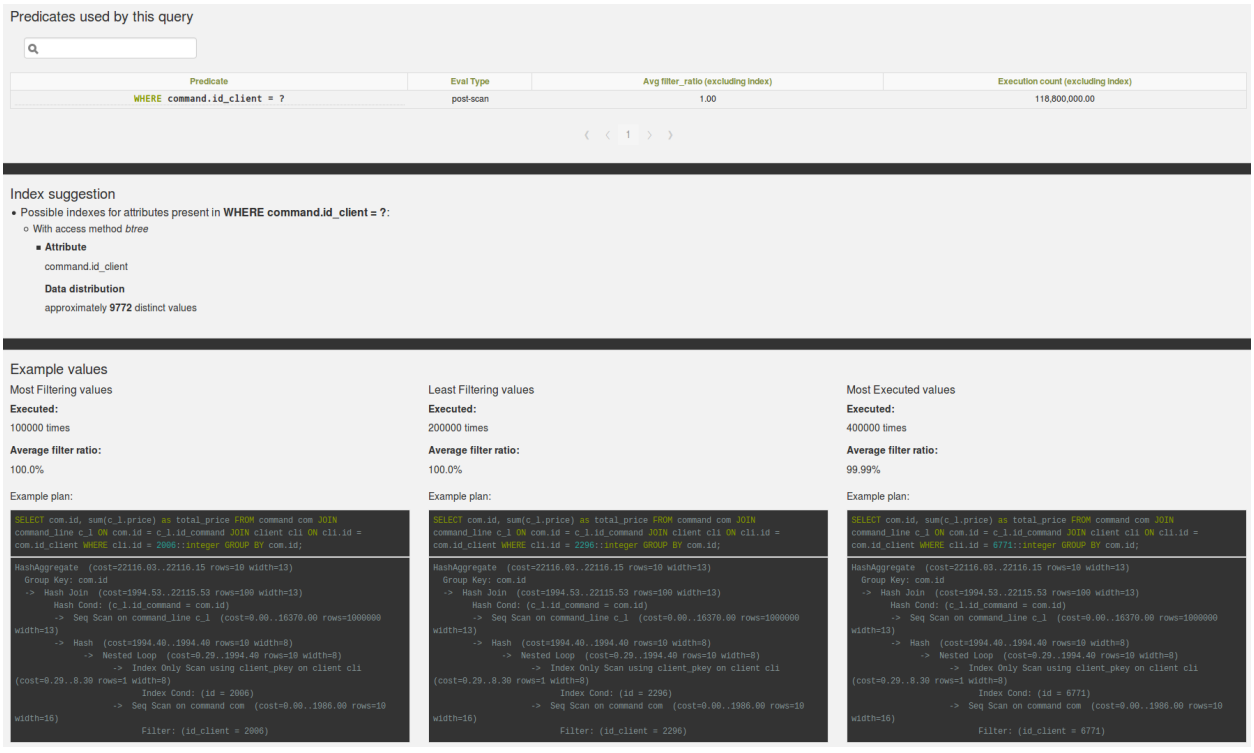
pg_qualstats is a PostgreSQL extension keeping statistics on predicates found in `WHERE` statements and `JOIN` clauses.

The goal of this extension is to allow the DBA to answer some specific questions, whose answers are quite hard to come by:

- what is the set of queries using this column ?
- what are the values this where clause is most often using ?
- do I have some significant skew in the distribution of the number of returned rows if use some value instead of one another ?
- which columns are often used together in a WHERE clause ?

### Where is it used in powa-web ?

If the extension is available, you should see a "list of quals" table on the query page, as well as explain plans for your query and a list of index suggestions:



From this list, you can then go on to the per-qual page.

### Installation

As seen in *Quickstart*, the PostgreSQL development packages should be available.

First, download and extract the latest release of *pg_qualstats*:

```
wget https://github.com/powa-team/pg_qualstats/archive/1.0.7.tar.gz -O pg_
↪qualstats-1.0.7.tar.gz
tar zxvf pg_qualstats-1.0.7.tar.gz
cd pg_qualstats-1.0.7
```

Then, compile the extension:

```
make
```

Then install the compiled package:

```
make install
```

Then you just have to declare the extension in the `postgresql.conf` file, like this :

```
shared_preload_libraries = 'pg_stat_statements,pg_qualstats'
```

Restart the PostgreSQL server to reload the libraries.

Connect to the server as a superuser and type:

```
CREATE EXTENSION pg_qualstats;
```

### Using with PoWA

If you want PoWA to handle this extension, you have to connect as a superuser on the database where you installed PoWA, and type:

```
SELECT powa_qualstats_register();
```

### Configuration

The following configuration parameters are available, in postgresql.conf:

**pg_qualstats.enabled:** Defaults to `true`. Enable pg_qualstats. Can be useful if you want to enable / disable it without restarting the server.

**pg_qualstats.max:** Defaults to `1000`. Number of entries to keep. As a rule of thumb, you should keep at least `pg_stat_statements.max` entries if `pg_qualstats.track_constants` is disabled, else it should be roughly equal to the number of queries executed during `powa.frequency` interval of time.

**pg_qualstats.track_pg_catalog:** Defaults to `false`. Determine if predicates on pg_catalog tables should be tracked too.

**pg_qualstats.resolve_oids:** Defaults to `false`. Determine if during predicates collection, the actual name of the objects should be stored alongside their OIDs. The overhead is quite non-negligible, since each entry will occupy 616 bytes instead of 168.

**pg_qualstats.track_constants:** Defaults to `true`. If true, each new value for each predicate will result in a new entry. Eg, `WHERE id = 3` and `WHERE id = 4` will results in two entries in pg_qualstats. If disabled, only one entry for `WHERE id = ?` will be kept. Turning this off drastically reduces the number of entries to keep, at the price of not getting any hindsight on most frequently used values.

**pg_qualstats.sample_rate:** (Used to be "sample_ratio") Defaults to `-1`, which means `1 / MAX_CONNECTIONS` The ratio of queries that should be sampled. 1 means sample every single query, 0 basically deactivates the feature, and -1 is automatically sized to `1/ MAX_CONNECTIONS`. For example, a sample_rate of `0.1` would mean one of out ten queries should be sampled.

### SQL Objects

The extension defines the following objects:

### pg_stat_kcache

pg_stat_kcache is a PostgreSQL extension gathering statistics on system metrics.

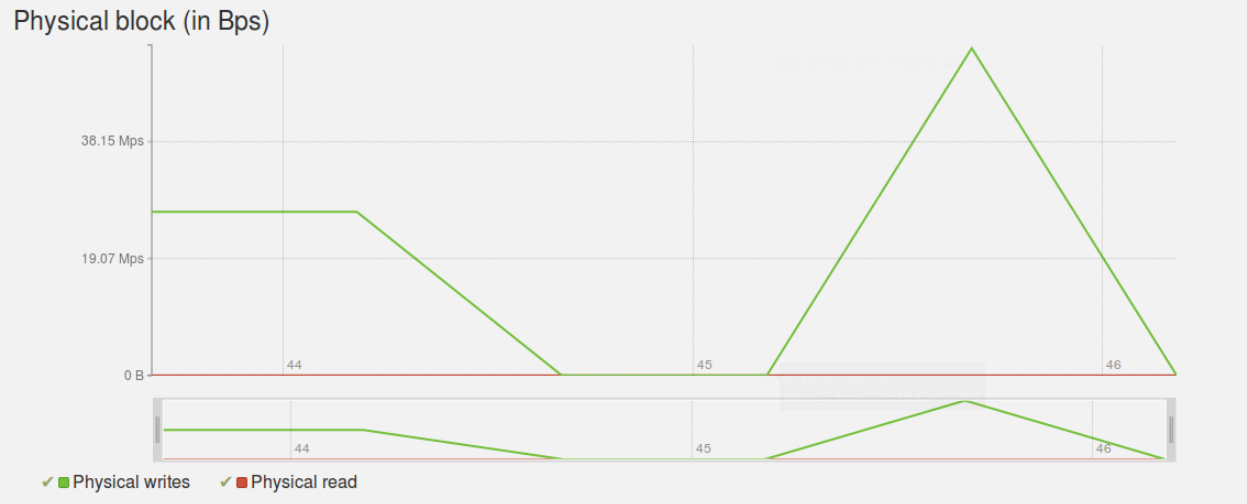
Thanks to this extension, the DBA can see how much resource each query, user and/or database is consuming. The resources are:

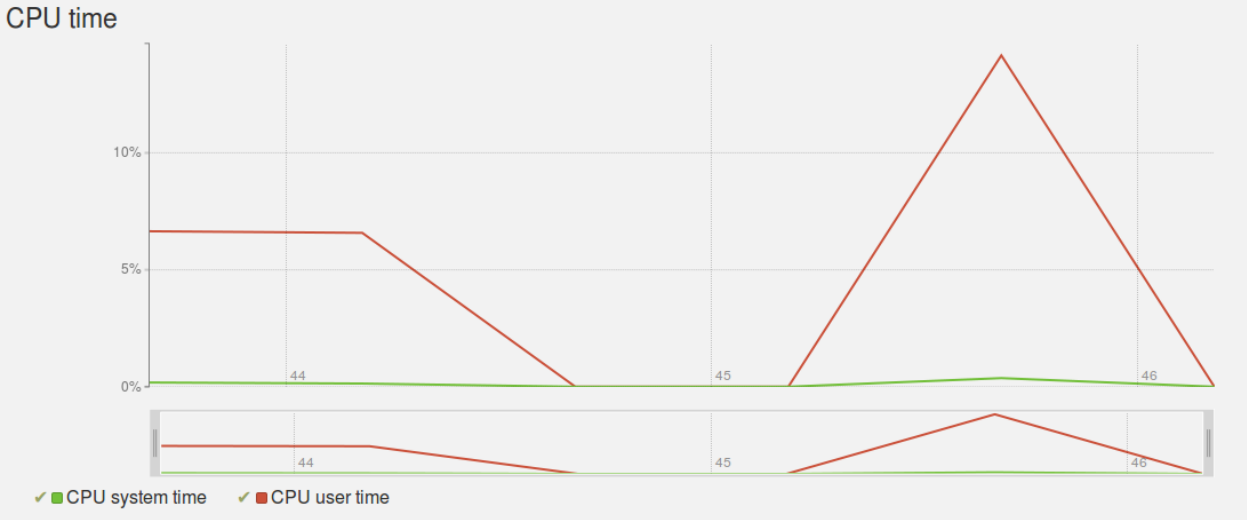
- CPU (user time and system time)
- Physical disk access (read and write)

Physical disk access are essential in calculating a real hit ratio (cached_reads/all_reads). Without this, we only have the shared_buffers' hit ratio, and some of the reads made by Postgres could be served by the system cache.

### Where is it used in powa-web ?

If the extension is available, you should see "Physical block" and "CPU time" graphs on the query page:



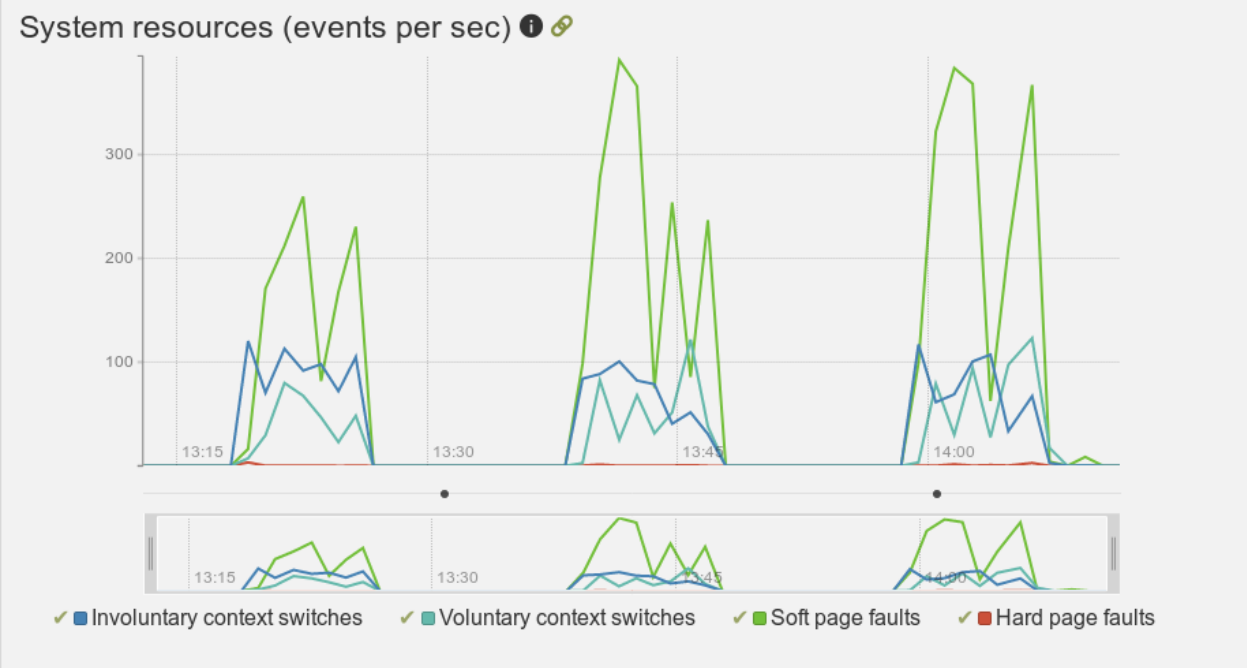



The **CPU time** metrics indicate the percentage of query runtime spent consuming either *user cpu time* or *system cpu time*.

The "Hit ratio" graph will also handle this extension, displaying the following metrics :

- **Shared buffers hit ratio**: percentage of blocks read from shared buffers (memory)
- **System cache hit ratio**: precentage of blocks read from the system cache (memory)
- **Disk hit ratio**: Percentage of blocks which needed a physical disk read

The rest of the available metrics will be displayed on a dedicated **System Resources** graph:




### Installation

pg_stat_kcache should work with any POSIX operating system. Therefore, it won't on Windows.

As seen in *Quickstart*, the PostgreSQL development packages should be available.

First, you need to download and extract the lastest release of pg_stat_kcache.

```
wget https://github.com/powa-team/pg_stat_kcache/archive/REL2_1_1.tar.gz -O␣
↪pg_stat_kcache-REL2_1_1.tar.gz
tar zxvf pg_stat_kcache-REL2_1_1.tar.gz
cd pg_stat_kcache-REL2_1_1
```

Then, compile the extension:

```
make
```

If everything goes fine, you will have this kind of output :

```
gcc -O0 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -
↪Wendif-labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -
↪fwrapv -fexcess-precision=standard -g -fpic -I. -I./ -I/home/rjuju/postgres/pgs/
↪postgresql-9.4.beta2/include/server -I/home/rjuju/postgres/pgs/postgresql-9.4.beta2/
↪include/internal -D_GNU_SOURCE -I/usr/include/libxml2   -c -o pg_stat_kcache.o pg_
↪stat_kcache.c
gcc -O0 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -
↪Wendif-labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -
↪fwrapv -fexcess-precision=standard -g -fpic -shared -o pg_stat_kcache.so pg_stat_
↪kcache.o -L/home/rjuju/postgres/pgs/postgresql-9.4.beta2/lib -L/usr/lib/x86_64-
↪linux-gnu  -Wl,--as-needed -Wl,-rpath,'/home/rjuju/postgres/pgs/postgresql-9.4.
↪beta2/lib',--enable-new-dtags
```

Then install the compiled file. This step has to be made with the user that has installed PostgreSQL. If you have used a package, it will be certainly be root. If so:

```
sudo make install
```

Else, sudo into the user that owns your PostgreSQL executables, and

```
make install
```

Then you just have to declare the extension in the `postgresql.conf` file, like this :

```
shared_preload_libraries = 'pg_stat_statements,pg_stat_kcache'
```

Restart the PostgreSQL server to reload the libraries.

Connect to the server as a superuser and type:

```
CREATE EXTENSION pg_stat_kcache ;
```

### Using with PoWA

If you want PoWA to handle this extension, you have to connect as a superuser on the database where you installed PoWA, and type:

```
SELECT powa_kcache_register();
```

### Configuration

pg_stat_kcache will retain as many query statistic as pg_stat_statements, so there's nothing to configure.

### Examples

### See Also

- *pg_stat_statements*
- *pg_qualstats*

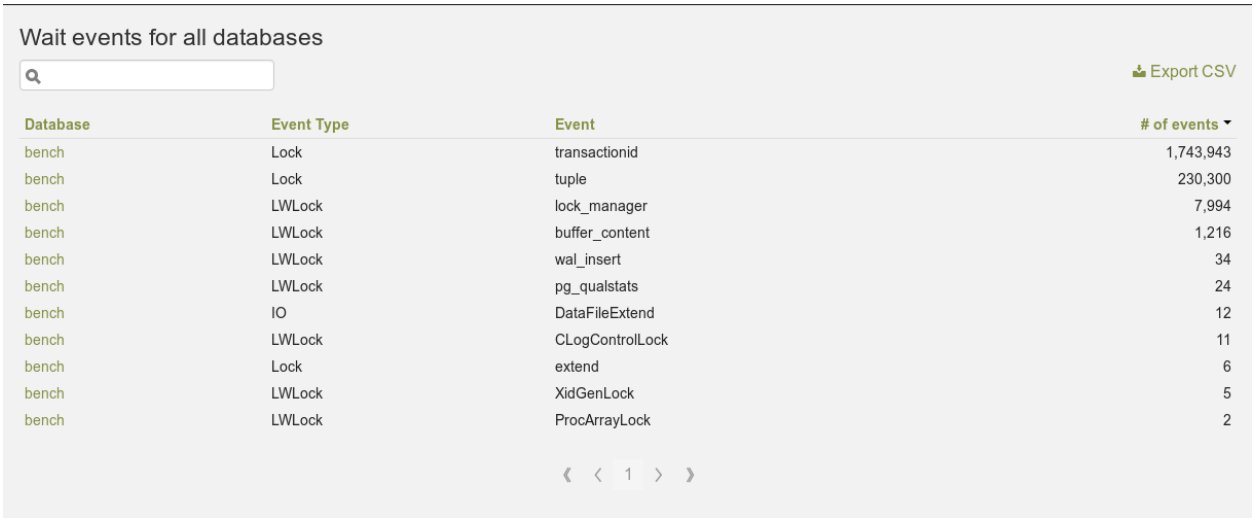
## pg_wait_sampling

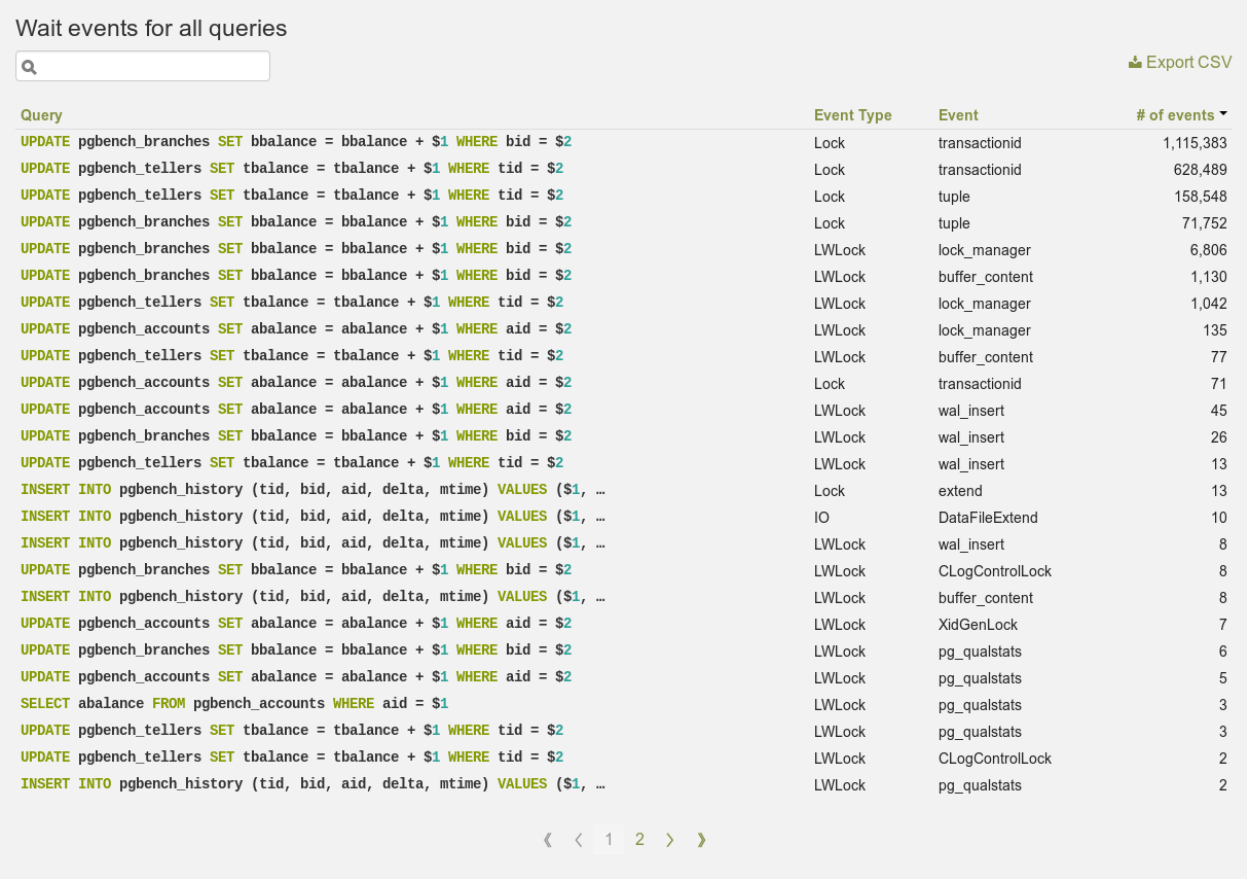
The pg_wait_sampling extension is devlopped by PostgresProfessional. It samples wait_events of all SQL queries executed on a given PostgreSQL server, providing **waits profile**, an accumulated view of wait events.

The **waits profile** is available in view called `pg_wait_sampling_profile`. This view contains one row for each distinct Process ID, wait event type, event and query ID.

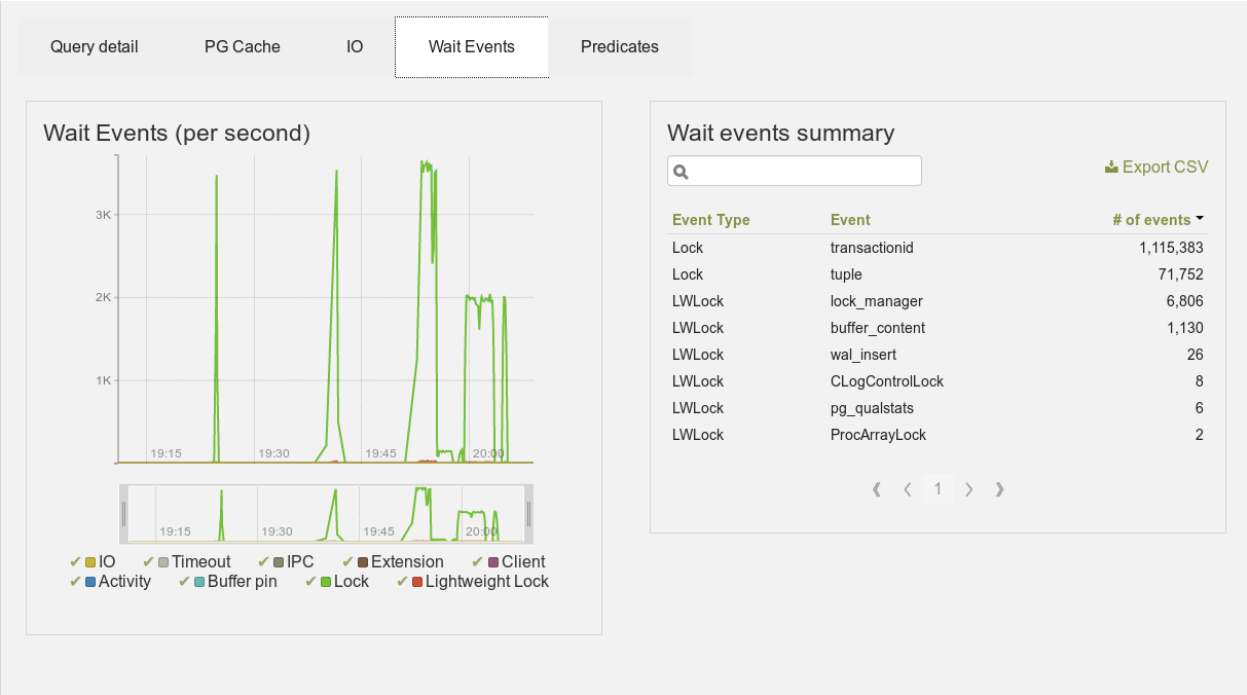
### Where is it used in powa-web ?

If the extension is available, you should see a "Wait events for all databases" table on the overview page and a "Wait events for all queries" table on the database page. Those tables report the list of reported wait events for the given period, either on the overall instance or on the database only.






On the query page, a “Wait Events” tab is available, where you’ll see both a graph of reported wait events, per type, and a table of all reported wait events, both for the given period.



### Installation

As seen in *Quickstart*, the PostgreSQL development packages should be available.

First, download and extract the latest release of pg_wait_sampling:

```
wget https://github.com/postgrespro/pg_wait_sampling/archive/v1.1.tar.gz -O␣
↪pg_wait_sampling-v1.1.tar.gz
tar zxvf pg_wait_sampling-v1.1.tar.gz
cd pg_wait_sampling-v1.1
```

Then, compile the extension:

```
make
```

Then install the compiled package:

```
make install
```

Then you just have to declare the extension in the `postgresql.conf` file, like this :

```
shared_preload_libraries = 'pg_stat_statements,pg_wait_sampling'
```

Restart the PostgreSQL server to reload the libraries.

Connect to the server as a superuser and type:

```
CREATE EXTENSION pg_wait_sampling;
```

### Using with PoWA

If you want PoWA to handle this extension, you have to connect as a superuser on the database where you installed PoWA, and type:

```
SELECT powa_wait_sampling_register();
```

### Configuration

For a complete description of the confirugration parameters, please refer to the official pg_wait_sampling documentation.

For PoWA needs, here are the important settings:

**pg_wait_sampling.profile_period:** Defaults to `10`. Period for profile sampling in milliseconds.

**pg_wait_sampling.profile_pid:** Defaults to `true`. Whether profile should be per pid. **Should be set to true for PoWA usage**.

**pg_wait_sampling.profile_queries:** Defaults to `false`. Whether profile should be per normalized query, as provided by *pg_stat_statements* extension. **Should be set to true for PoWA usage**.

### See Also

- List of wait events in PostgreSQL

### pg_track_settings

The pg_track_settings extension is a small SQL-only extension. Its purpose is to keep track of configuration changes happening on your instances. You can see more details of how to use this extension on a presentation article.

This extension will record any change happening in

- the main configuration settings (as configured in *postgresql.conf* or with *ALTER SYSTEM* for instance), as reported by the **pg_settings** view.
- the per-user and/or per-database settings (*ALTER ROLE ... SET*, *ALTER DATABASE ... SET* and *ALTER ROLE ... IN DATABASE SET*), as reported by the **pg_db_role_setting** table
- PostgreSQL restart, using the **pg_postmaster_start_time()** function

when the **snapshot function** is called (or the **functions** starting from version 2.0.0).

---

**Note:** If the user running the snapshot function has a per-user and/or a per-database settings, this setting will "hide" the regular value in *pg_setting*, so keep this restriction in mind when investigatin the extension reports.

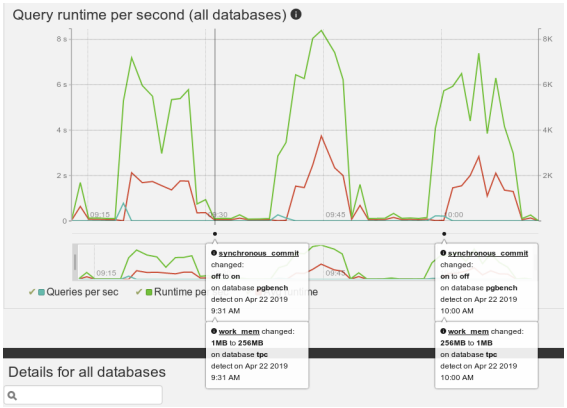
---

All versions are compatible with PoWA with the standalone setup. Since version 2.0.0, pg_track_settings is compatible with the *Remote setup* added in PoWA 4.

#### Where is it used in powa-web ?

If the extension is properly configured, you should see a timeline widget, placed between each graph and its overview, displaying any kind of recorded change if any was detected in the currently selected time interval. This list will be filtered by the database currently displayed if the current page is displaying a specific database. This timeline will be displayed on every graph of the page, to easily check if this change had any visible impact.

Details of the changes will be displayed on mouseover. You can click on any event on the timeline to make the event stay displayed, and draw a vertical line on the underlying graph.




#### Installation

As seen in *Quickstart*, the PostgreSQL development packages should be available.

First, download and extract the latest release of pg_track_settings:

```
wget https://github.com/rjuju/pg_track_settings/archive/2.0.0.tar.gz -O pg_
↪track_settings-2.0.0.tar.gz
```

```
tar zxvf pg_track_settings-2.0.0.tar.gz
cd pg_track_settings-2.0.0
```

Since it's an SQL-only extension, there's no need to compile anything. You just need to install the package:

```
make install
```

No specific configuration or PostgreSQL restart is needed. Simply connect on the PoWA database as a superuser and type:

```
CREATE EXTENSION pg_track_settings;
```

---

**Note:** If you're installing a *Remote setup* configuration, then you need **at least the version 2.0.0** of the extension. It also has to be intalled:

- on the dedicated powa database of the **repository server**
- on the dedicated powa database of all the **remote servers** for which you want to track the configuration changes

---

#### Using with PoWA

If you want PoWA to handle this extension, you have to connect as a superuser on the database where you installed PoWA, and type:

```
SELECT powa_wait_sampling_register();
```

All those extensions have to be installed on the dedicated powa database of the monitored server.

---

**Note:** pg_track_settings has to be also be installed on the dedicated repository server if *Remote setup* configuration is used.

---

### 1.5.5 HypoPG

HypoPG is a stat extension, but it's a useful extension to take full advantage of all the PoWA features.

HypoPG allows you to create hypothetical indexe. A hypothetical index is an index that doesn't exists on disk. It's therefore almost instant to create and doesn't add any IO cost, whether at creation time or at maintenance time. The goal is obviously to check if an index is useful before spending too much time, I/O and disk space to create it.

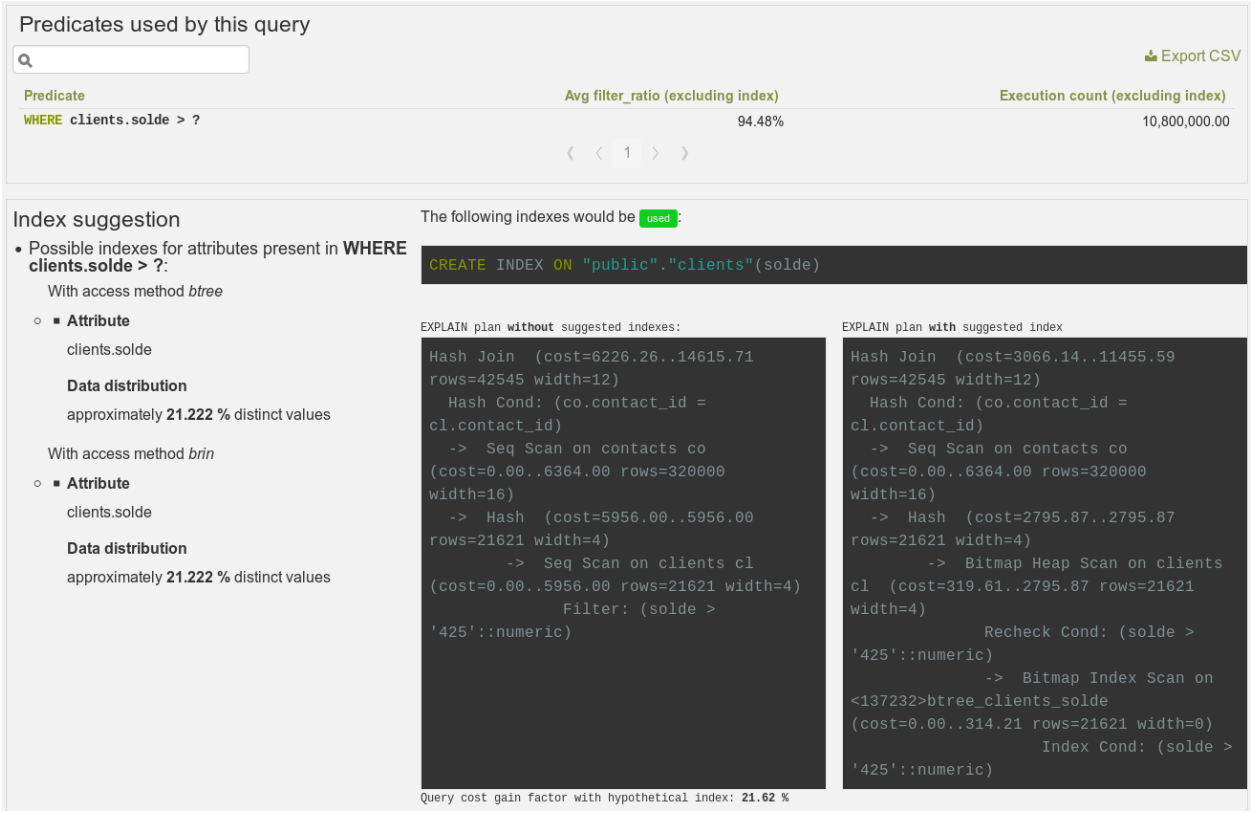
With this extension, you can create hypothetical indexes, and then with EXPLAIN check if PostgreSQL would use them or not.

#### Where is it used in powa-web ?

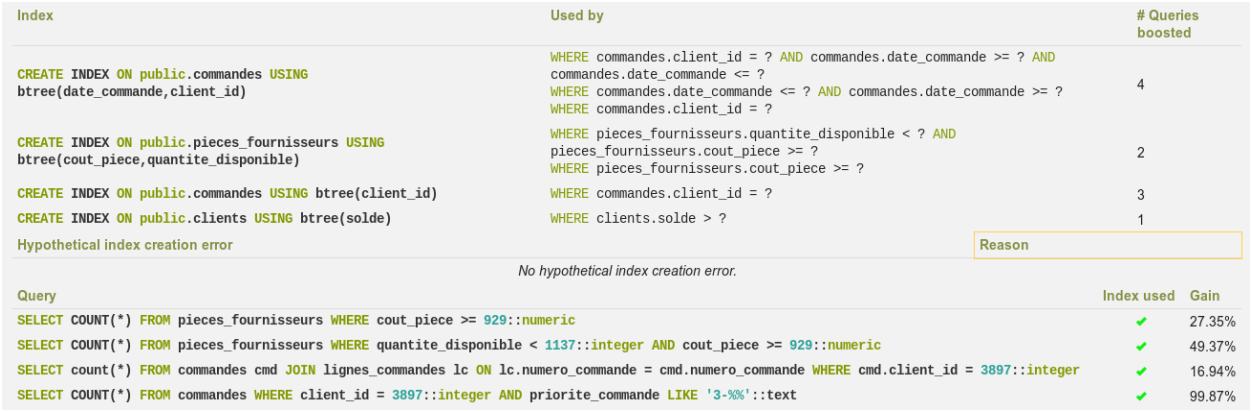
If *pg_qualstats* is configured, PoWA will be able to detect missing indexes, either per-query or for the **whole workkload of a database**!

When PoWA shows suggestion of missing indexes, if HypoPG is available **on the target database** (of the **remote server** if the *Remote setup* mode is used), it'll also try to create a hypothetical index for each suggested index, and show you if PostgreSQL would use it or not.

This can be seen on the per-query page, in the **Predicates** tab:




And on the database page, if you use the "Optimize this database" feature:




## Installation

As seen in *Quickstart*, the PostgreSQL development packages should be available.

First, you need to download and extract the lastest release of hypopg.

```
wget https://github.com/hypopg/hypopg/archive/1.1.2.tar.gz -O hypopg-1.1.2.
↪tar.gz
tar zxvf hypopg-1.1.2.tar.gz
cd hypopg-1.1.2
```

Then, compile the extension:

```
make
```

Then install the compiled file. This step has to be made with the user that has installed PostgreSQL. If you have used a package, it will be certainly be root. If so:

```
sudo make install
```

Else, sudo into the user that owns your PostgreSQL executables, and

```
make install
```

No specific configuration or PostgreSQL restart is needed.

Connect as a superuser on each database of each server you want to be able to use hypopg on, and type:

```
CREATE EXTENSION hypopg ;
```

#### See Also

- *pg_qualstats*
- Official documentation

## 1.6 Impact on performances

Using PoWA can have a small negative impact on your PostgreSQL server performances. It is hard to evaluate precisely this impact, as it can come from different parts.

First of all, you need to activate at least pg_stat_statements extension, and possibly the other supported *Stats Extensions* of your choice. Those extensions can slow down your instance, depending on how you configuraiton them.

If you don't use the *Remote setup* mode, the data will be stored locally on a regular basis. Depending on the snapshot frequency, the overhead could be important. You also have to consider disk usage, which will impact at least the backups.

Using the UI will also run queries on your databases. With the *Remote setup* mode, there should be very few queries run on the target databases though.

## 1.7 Support

### 1.7.1 Community Support

You can join directly the developer team on the #powa channel of the freenode IRC network.

To report an issue, please use the bug tracking system in the github page of the underlying project:

- https://github.com/powa-team/powa/issues for a general issue
- https://github.com/powa-team/powa-web/issues for an issue on the UI
- https://github.com/powa-team/powa-archivist/issues for an issue on the background worker
- https://github.com/powa-team/powa-collector/issues for an issue on the collector daemon
- https://github.com/powa-team/pg_qualstats/issues for an issue on pg_qualstats

- https://github.com/powa-team/pg_stat_kcache/issues for an issue on pg_stat_kcache
- https://github.com/postgrespro/pg_wait_sampling/issues for an issue on pg_wait_wampling
- https://github.com/rjuju/pg_track_settings/issues for an issue on pg_track_settings
- https://github.com/HypoPG/hypopg/issues for an issue on HypoPG

## 1.8 Release Notes

The release notes of each component (internal or external) are available at:

- powa-archivist: https://github.com/powa-team/powa-archivist/releases
- powa-web: https://github.com/powa-team/powa-web/releases
- powa-collector: https://github.com/powa-team/powa-collector/releases
- HypoPG: https://github.com/HypoPG/hypopg/releases
- pg_qualstats: https://github.com/powa-team/pg_qualstats/releases
- pg_stat_kcache: https://github.com/powa-team/pg_stat_kcache/releases
- pg_wait_sampling: https://github.com/postgrespro/pg_wait_sampling/releases
- pg_track_settings: https://github.com/rjuju/pg_track_settings/releases

You can also consult these page for the major version changlog:

### 1.8.1 What's new in PoWA 3.0.0

**December 7, 2015**

#### Better predicate analyzer

The pg_qualstats (https://github.com/powa-team/pg_qualstats) extension stores new counters. It's now possible to know the most executed predicates in relation to all the related queries. It also tracks non-normalized queries so that it's possible to execute an EXPLAIN of any query tracked by pg_stat_statements.

#### Database global optimization

PoWA is now able to use statistics about every predicate used by any query executed on a database to suggest the smallest index set that optimizes every one of those predicates.

In particular, the heuristics place heavy emphasis in consolidating many indexes into one by giving preference to definitions spanning multiple columns. This can provide new information about the actual load and correlation between predicates that are traditionally hard to discover for the DBA.

#### Index suggestion check

Thanks to the HypoPG (https://github.com/hypopg/hypoopg) extension, the benefits of the suggested index creations can automatically be checked by running the queries against hypothetical indexes. You can see instantly if the suggested index is relevant and how much it'll improve the query.

#### Documentation

- Complete user documentation available at http://powa.readthedocs.io/

#### Backward compatibility

- PoWA 2.0 and later is NOT COMPATIBLE with PostgreSQL 9.3. If you're using PoWA with PostgreSQL 9.3, you can either keep PoWA 1.2 or upgrade to PostgreSQL 9.4 and switch to PoWA 3.0.0.

### 1.8.2 What's new in PoWA 2.0

**March 2, 2015**

#### New User Interface

- The web interface is now a seprate module called `powa-web`
- Complete rewite of the previous HTML UI
- We dropped mojolicious and use Tornado instead
- New Bar Graph
- New configuration view
- New index suggestion widget
- New physical resource consumption graphs
- Pie Charts
- Histogramm for qual constants values
- Better Global Query Chart
- Breadcrumbs
- Check PoWA installation on login
- Python 2.6, 2.7 and 3.4 compatibility

#### New Stat sources

- The core engine is now a seprate module called `powa-archivist`
- Integration of pg_qualstats
- Integration of pg_stat_kcache

#### Documentation

- Complete user documentation available at http://powa.readthedocs.io/

#### Backward compatibility

- PoWA 2.0 and later is NOT COMPATIBLE with PostgreSQL 9.3. If you're using PoWA with PostgreSQL 9.3, you can either keep PoWA 1.2 or upgrade to PostgreSQL 9.4 and switch to PoWA 2.0.

## 1.9 Contributing

POWA is an open project available under the PostgreSQL License. Any contribution to build a better tool is welcome.

### 1.9.1 Talk

If you have ideas or feature requests, please post them to our general bug tracker: https://github.com/powa-team/powa/issues

You can also join the **#powa** IRC channel on freenode server.

### 1.9.2 Test

If you've found a bug, please refer to *Support* page to see how to report it.

### 1.9.3 Code

PoWA is composed of multiples tools:

- a background worker, see *PoWA archivist*
- a collector daemon, see *PoWA-collector*
- stats extensions, see *Stats Extensions*
- a UI, see *PoWA-web*
- external extensions:
  - *HypoPG*
  - *pg_wait_sampling*
  - *pg_track_settings*