pgwatch2

May 17, 2022

Contents:

1 Introduction **1**

1.1 Quick start with Docker . . . 1
1.2 Typical "pull" architecture . . . 2
1.3 Typical "push" architecture . . . 2

2 Project background **5**

2.1 Project feedback . . . 5

3 List of main features **7**

4 Advanced features **9**

4.1 Patroni support . . . 9
4.2 Log parsing . . . 10
4.3 PgBouncer support . . . 10
4.4 Pgpool-II support . . . 11
4.5 Prometheus scraping . . . 12
4.6 AWS / Azure / GCE support . . . 12

5 Components **13**

5.1 The metrics gathering daemon . . . 13
5.2 Configuration store . . . 13
5.3 Metrics storage DB . . . 14
5.4 The Web UI . . . 14
5.5 Metrics representation . . . 14
5.6 Component diagram . . . 15
5.7 Component reuse . . . 15

6 Installation options **17**

6.1 Config DB based operation . . . 17
6.2 File based operation . . . 17
6.3 Ad-hoc mode . . . 17
6.4 Prometheus mode . . . 18

7 Installing using Docker **19**

7.1 Simple setup steps . . . 19
7.2 More future proof setup steps . . . 20
7.3 Available Docker images . . . 20

7.4 Building custom Docker images . . . 21
7.5 Interacting with the Docker container . . . 21
7.6 Ports used . . . 22
7.7 Docker Compose . . . 22

8 Custom installation **23**
8.1 Config DB based setup . . . 23
8.2 YAML based setup . . . 28
8.3 Using InfluxDB for metrics storage . . . 28

9 Preparing databases for monitoring **31**
9.1 Effects of monitoring . . . 31
9.2 Basic preparations . . . 31
9.3 Rolling out helper functions . . . 32
9.4 Automatic rollout of helpers . . . 33
9.5 PL/Python helpers . . . 33
9.6 Notice on using metric fetching helpers . . . 34
9.7 Running with developer credentials . . . 34
9.8 Different *DB types* explained . . . 35

10 Monitoring managed cloud databases **37**
10.1 Google Cloud SQL for PostgreSQL . . . 37
10.2 Amazon RDS for PostgreSQL . . . 38
10.3 Azure Database for PostgreSQL . . . 38
10.4 Aiven for PostgreSQL . . . 38

11 Metric definitions **39**
11.1 Defining custom metrics . . . 40
11.2 Metric attributes . . . 41
11.3 Column attributes . . . 42
11.4 Adding metric fetching helpers . . . 42

12 The Admin Web UI **43**
12.1 Web UI security . . . 44
12.2 Exposing the component logs . . . 44

13 Dashboarding and alerting **45**
13.1 Grafana intro . . . 45
13.2 Alerting . . . 45

14 Sizing recommendations **47**

15 Technical details **49**

16 Security aspects **51**
16.1 General security information . . . 51
16.2 Launching a more secure Docker container . . . 52

17 Long term installations **53**
17.1 Keeping inventory in sync . . . 53
17.2 Updating the pgwatch2 collector . . . 53
17.3 Metrics maintenance . . . 54
17.4 Dashboard maintenance . . . 54
17.5 Storage monitoring . . . 54

18 Upgrading 55
- 18.1 Updating to a newer Docker version . . . 55
- 18.2 Updating without Docker . . . 56
- 18.3 Updating Grafana . . . 56
- 18.4 Updating Grafana dashboards . . . 56
- 18.5 Updating the config / metrics DB version . . . 56
- 18.6 Updating the pgwatch2 schema . . . 57
- 18.7 Updating the metrics collector . . . 57
- 18.8 Updating the Web UI . . . 57
- 18.9 Updating metric definitions . . . 58

19 Kubernetes 59

20 Indices and tables 61

CHAPTER 1

Introduction

pgwatch2 is a flexible PostgreSQL-specific monitoring solution, relying on Grafana dashboards for the UI part. It supports monitoring of almost all metrics for Postgres versions 9.0 to 13 out of the box and can be easily extended to include custom metrics. At the core of the solution is the metrics gathering daemon written in Go, with many options to configure the details and aggressiveness of monitoring, types of metrics storage and the display the metrics.

1.1 Quick start with Docker

For the fastest setup experience Docker images are provided via Docker Hub (if new to Docker start here). For custom setups see the *Custom installations* paragraph below or turn to the pre-built DEB / RPM / Tar packages on the Github Releases page.

Launching the latest pgwatch2 Docker image with built-in InfluxDB metrics storage DB:

```
# run the latest Docker image, exposing Grafana on port 3000 and the administrative␣
↪web UI on 8080
docker run -d -p 3000:3000 -p 8080:8080 -e PW2_TESTDB=true --name pw2 cybertec/
↪pgwatch2
```

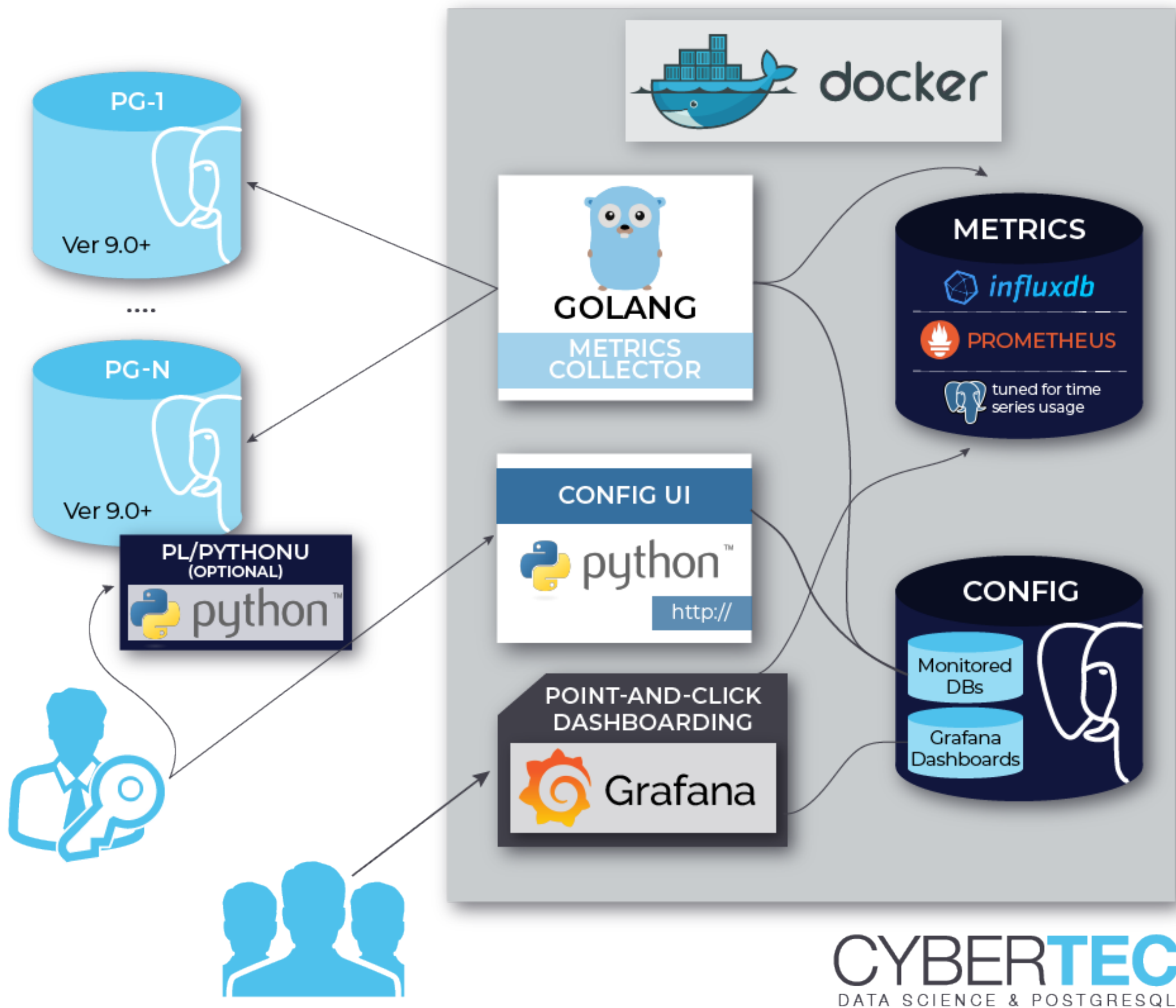
After some minutes you could for example open the "DB overview" dashboard and start looking at metrics in Grafana. For defining your own dashboards or making changes you need to log in as admin (default user/password: admin/pgwatch2admin).

NB! If you don't want to add the "test" database (the pgwatch2 configuration DB holding connection strings to monitored DBs and metric definitions) to monitoring, remove the PW2_TESTDB env variable.

Also note that for long term production usage with Docker it's highly recommended to use separate *volumes* for each pgwatch2 component - see *here* for a better launch example.

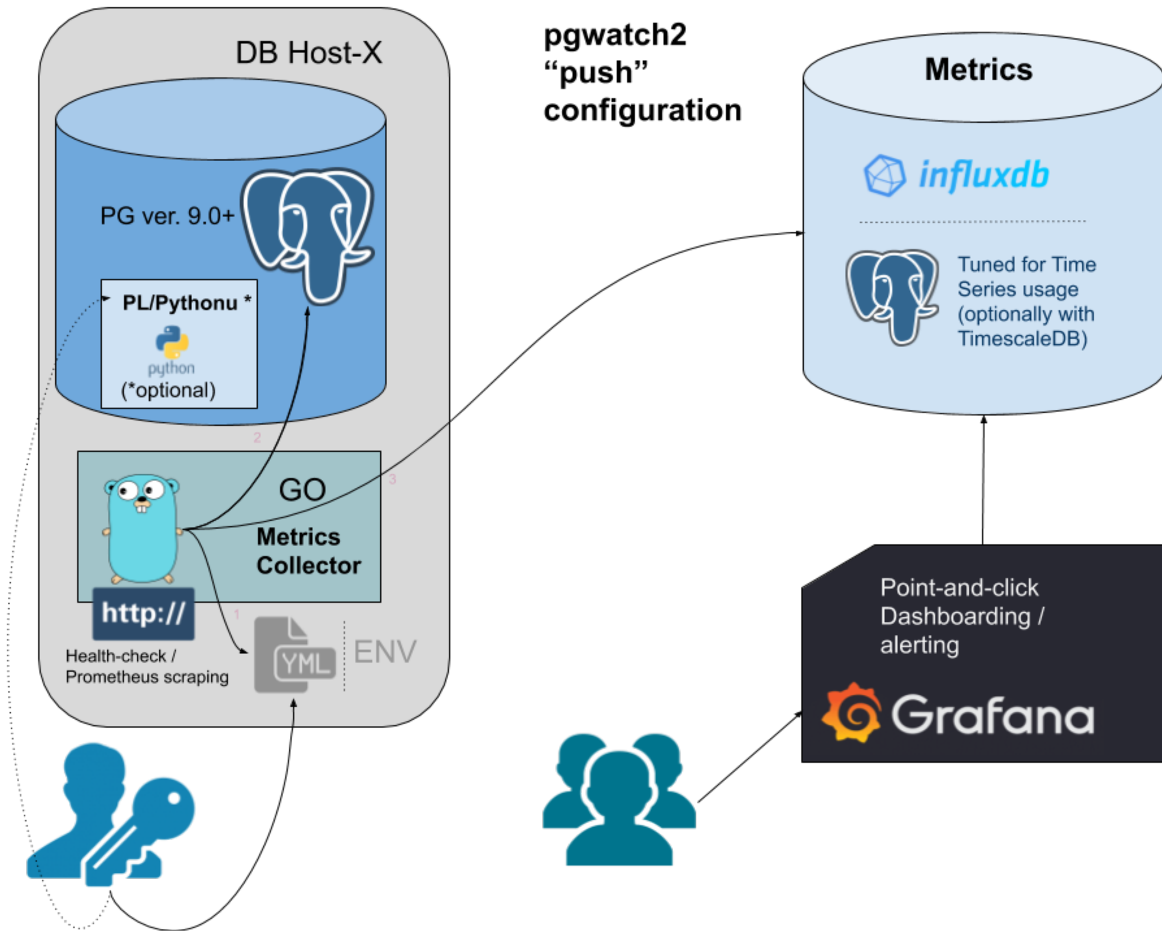
1.2 Typical "pull" architecture

To get an idea how pgwatch2 is typically deployed a diagram of the standard Docker image fetching metrics from a set of Postgres databases configured via a configuration DB:



1.3 Typical "push" architecture

A better fit for very dynamic (Cloud) environments might be a more de-centralized "push" approach or just exposing the metrics over a port for remote scraping. In that case the only component required would be the pgwatch2 metrics collection daemon.


DB Host-X
PG ver. 9.0+
PL/Pythonu *
python
(*optional)
GO
Metrics
Collector
http://
Health-check /
Prometheus scraping
YML
ENV
pgwatch2
"push"
configuration
Metrics
influxdb
Tuned for Time
Series usage
(optionally with
TimescaleDB)
Point-and-click
Dashboarding /
alerting
Grafana

CHAPTER 2

Project background

The pgwatch2 project got started back in 2016 and released in 2017 initially for internal monitoring needs at Cybertec as all the Open Source PostgreSQL monitoring tools at the time had various limitations like being too slow and invasive to set up or providing a fixed set of visuals and metrics.

For more background on the project motivations and design goals see the original series of blogposts announcing the project and the following feature updates released approximately twice per year.

Cybertec also provides commercial 9-to-5 and 24/7 support for pgwatch2.

- Project announcement
- Implementation details
- Feature pack 1
- Feature pack 2
- Feature pack 3
- Feature pack 4
- Feature pack 5
- Feature pack 6
- Feature pack 7

2.1 Project feedback

For feature requests or troubleshooting assistance please open an issue on project's Github page.

CHAPTER 3

List of main features

- Non-invasive setup on PostgreSQL side - no extensions nor superuser rights are required for the base functionality so that even unprivileged users like developers can get a good overview of database activities without any hassle
- Lots of preset metric configurations covering all performance critical PostgreSQL internal Statistics Collector data
- Intuitive metrics presentation using a set of predefined dashboards for the very popular Grafana dashboarding engine with optional alerting support
- Easy extensibility of metrics which are defined in pure SQL, thus they could also be from the business domain
- Many metric data storage options - PostgreSQL, PostgreSQL with the compression enabled TimescaleDB extension, InfluxDB, Graphite or Prometheus scraping
- Multiple deployment options - PostgreSQL configuration DB, YAML or ENV configuration, supporting both "push" and "pull" models
- Possible to monitoring all, single or a subset (list or regex) of databases of a PostgreSQL instance
- Global or per DB configuration of metrics and metric fetching intervals and optionally also times / days
- Kubernetes/OpenShift ready with sample templates and a Helm chart
- PgBouncer, Pgpool2, AWS RDS and Patroni support with automatic member discovery
- Internal health-check API (port 8081 by default) to monitor metrics gathering status / progress remotely
- Built-in security with SSL connections support for all components and passwords encryption for connect strings
- Very low resource requirements for the collector even when monitoring hundreds of instances
- Capabilities to go beyond PostgreSQL metrics gathering with built-in log parsing for error detection and OS level metrics collection via PL/Python "helper" stored procedures
- A *Ping mode* to test connectivity to all databases under monitoring

CHAPTER 4

Advanced features

Over the years the core functionality of fetching metrics from a set of plain Postgres DB-s has been extended in many ways to cover some common problem areas like server log monitoring and supporting monitoring of some other popular tools often used together with Postgres, like the PgBouncer connection pooler for example.

4.1 Patroni support

Patroni is a popular Postgres specific HA-cluster manager that makes node management simpler than ever, meaning that everything is dynamic though - cluster members can come and go, making monitoring in the standard way a bit tricky. But luckily Patroni cluster members information is stored in a DCS (Distributed Consensus Store), like *etcd*, so it can be fetched from there periodically.

When 'patroni' is selected as *DB type* then the usual Postgres host/port fields should be left empty ("dbname" can still filled if only a specific single database is to be monitored) and instead "Host config" JSON field should be filled with DCS address, type and scope (cluster name) information. A sample config (for Config DB based setups) looks like:

```
{
  "dcs_type": "etcd",
  "dcs_endpoints": ["http://127.0.0.1:2379"],
  "scope": "batman",
  "namespace": "/service/"
}
```

For YAML based setups an example can be found from the instances.yaml file.

NB! If Patroni is powered by *etcd*, then also username, password, ca_file, cert_file, key_file optional security parameters can be defined - other DCS systems are currently only supported without authentication.

Also if you don't use the standby nodes actively for queries then it might make sense to decrease the volume of gathered metrics and to disable the monitoring of such nodes with the "Master mode only?" checkbox (when using the Web UI) or with *only_if_master=true* if using a YAML based setup.

4.2 Log parsing

As of v1.7.0 the metrics collector daemon, when running on a DB server (controlled best over a YAML config), has capabilities to parse the database server logs for errors. Out-of-the-box it will though only work when logs are written in **CVSLOG** format. For other formats user needs to specify a regex that parses out named groups of following fields: *database_name*, *error_severity*. See here for an example regex.

NB! Note that only the event counts are stored, no error texts, usernames or other infos! Errors are grouped by severity for the monitored DB and for the whole instance. The metric name to enable log parsing is "server_log_event_counts". Also note that for auto-detection of log destination / setting to work, the monitoring user needs superuser / pg_monitor privileges - if this is not possible then log settings need to be specified manually under "Host config" as seen for example here.

Sample configuration if not using CSVLOG logging:

On Postgres side (on the monitored DB)

```
# Debian / Ubuntu default log_line_prefix actually
log_line_prefix = '%m [%p] %q%u@%d '
```

YAML config (recommended when "pushing" metrics from DB nodes to a central metrics DB)

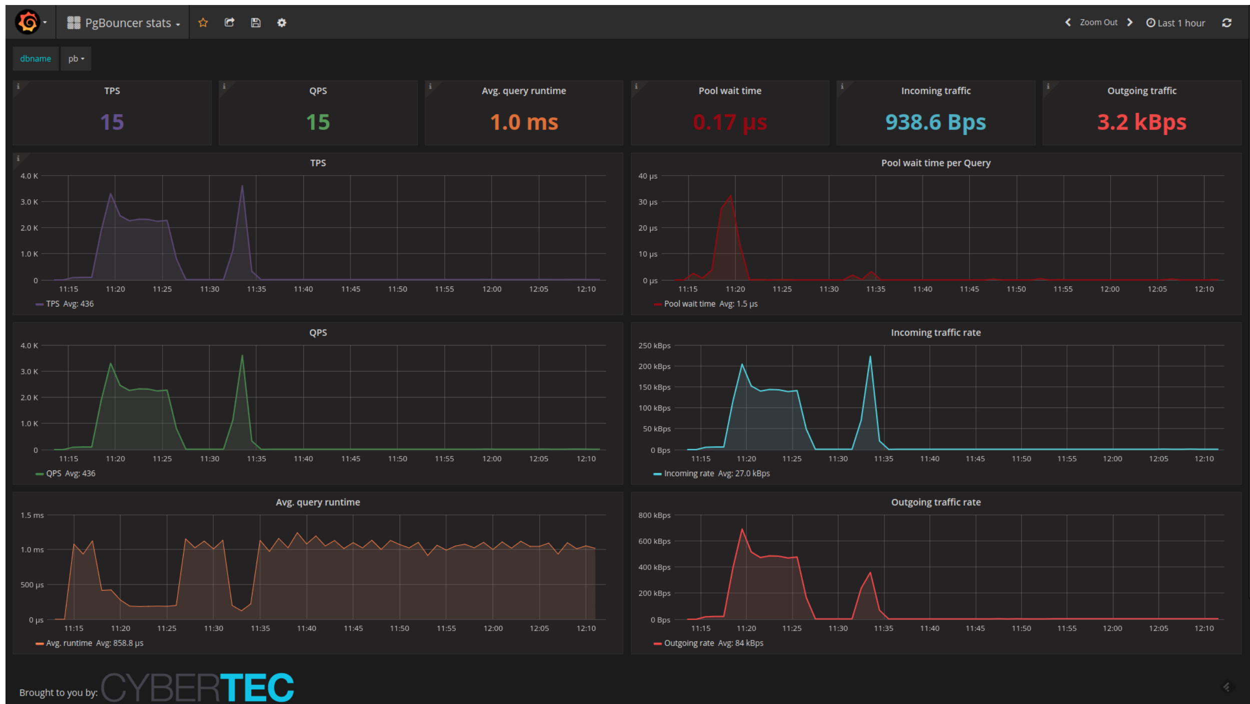
```
## logs_glob_path is only needed if the monitoring user is cannot auto-detect it (i.e.
↪ not a superuser / pg_monitor role)
# logs_glob_path:
logs_match_regex: '^(?P<log_time>.*) \[(?P<process_id>\d+)\] (?P<user_name>.*)@(?P
↪<database_name>.*?) (?P<error_severity>.*?): '
```

NB! For log parsing to work the metric **server_log_event_counts** needs to be enabled or a *preset config* including it used - like the "full" preset.

4.3 PgBouncer support

Pgwatch2 also supports collecting internal statistics from the PgBouncer connection pooler, via the built-in special "pgbouncer" database and the `SHOW STATS` command. To enable it choose the according *DB Type*, provide connection info to the pooler port and make sure the **pgbouncer_stats** metric or "pgbouncer" preset config is selected for the host. Note that for the "DB Name" field you should insert not "pgbouncer" (although this special DB provides all the statistics) but the real name of the pool you wish to monitor or leave it empty to track all pools. In latter case individual pools will be identified / separated via the "database" tag.

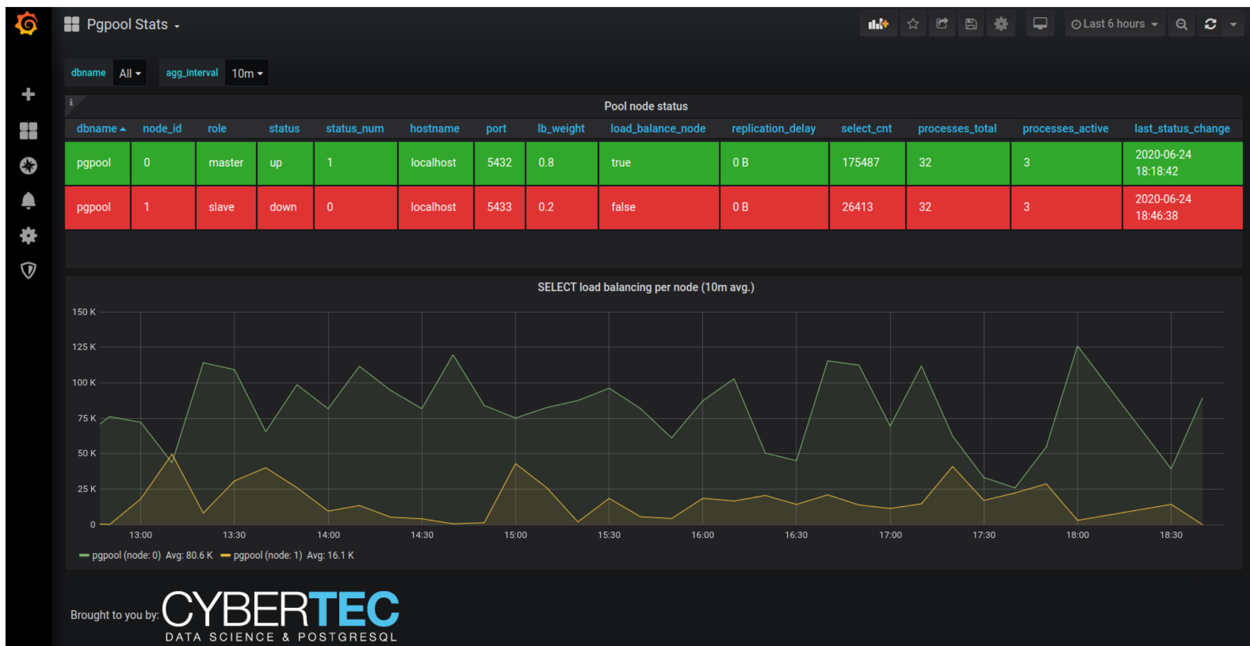
There's also a built-in Grafana dashboard for PgBouncer data, looking like that:



4.4 Pgpool-II support

Quite similar to PgBouncer, also Pgpool offers some statistics on pool performance and status, which might be of interest especially if using the load balancing features. To enable it choose the according *DB Type*, provide connection info to the pooler port and make sure the **pgpool_stats** metric / preset config is selected for the host.

The built-in Grafana dashboard for Pgpool data looks something like that:



dbname	node_id	role	status	status_num	hostname	port	lb_weight	load_balance_node	replication_delay	select_cnt	processes_total	processes_active	last_status_change
pgpool	0	master	up	1	localhost	5432	0.8	true	0 B	175487	32	3	2020-06-24 18:18:42
pgpool	1	slave	down	0	localhost	5433	0.2	false	0 B	26413	32	3	2020-06-24 18:46:38

4.5 Prometheus scraping

pgwatch2 was originally designed with direct metrics storage in mind, but later also support for externally controlled Prometheus scraping was added. Note that currently though the storage modes are exclusive, i.e. when you enable the Promotheus endpoint (default port 9187) there will be no direct metrics storage.

To enable the scraping endpoint set `--datastore=prometheus` and optionally also `--prometheus-port`, `--prometheus-namespace`, `--prometheus-listen-addr`. Additionally note that you still need to specify some metrics config as usually - only metrics with interval values bigger than zero will be populated on scraping.

NB! Currently a few built-in metrics that require some state to be stored between scrapes, e.g. the "change_events" metric, will currently be ignored. Also non-numeric data columns will be ignored! Tag columns will be preserved though as Prometheus "labels".

4.6 AWS / Azure / GCE support

Due to popularity of various managed PostgreSQL offerings there's also support for some managed options in sense of *Preset Configs*, that take into account the fact that on such platforms you get a limited user that doesn't have access to all metrics or some features have just been plain removed. Thus to reduce server log errors and save time on experimenting there are following presets available:

- **aws** - for standard AWS RDS managed PostgreSQL databases
- **aurora** - for AWS Aurora managed PostgreSQL service
- **azure** - for Azure Database for PostgreSQL managed databases
- **gce** - for Google Cloud SQL for PostgreSQL managed databases

CHAPTER 5

Components

The main development idea around pgwatch2 was to do the minimal work needed and not to reinvent the wheel - meaning that pgwatch2 is mostly just about gluing together already some proven pieces of software for metrics storage and using Grafana for dashboarding. So here a listing of components that can be used to build up a monitoring setup around the pgwatch2 metrics collector. Note that most components are not mandatory and for tasks like metrics storage there are many components to choose from.

5.1 The metrics gathering daemon

The metrics collector, written in Go, is the only mandatory and most critical component of the whole solution. The main task of the pgwatch2 collector / daemon is pretty simple - reading the configuration and metric defintions, fetching the metrics from the configured databases using the configured connection info and finally storing the metrics to some other database, or just exposing them over a port for scraping in case of Prometheus mode.

5.2 Configuration store

The configuration says which databases, how often and with which metrics (SQL-s queries) are to be gathered. There are 3 options to store the configuration:

- A PostgreSQL database holding a simple schema with 5 tables.
- File based approach - YAML config file(s) and SQL metric definition files.
- Purely ENV based setup - i.e. an "ad-hoc" config to monitor a single database or the whole instance. Bascially just a connect string (JDBC or Libpq type) is needed which is perfect for "throwaway" and Cloud / container usage.

5.3 Metrics storage DB

Many options here so that one can for example go for maximum storage effectiveness or pick something where they already know the query language:

- InfluxDB Time Series Database

  InfluxDB is specifically designed for storing metrics so it's disk footprint is much smaller compared to PostgreSQL but downsides are the limited query capabilities and higher hardware (CPU / RAM) requirements - see the *Sizing recommendations* chapter for more details.

- PostgreSQL - world's most advanced Open Source RDBMS

  Postgres storage is based on the JSONB datatype so minimally version 9.4+ is required, but for bigger setups where partitioning is a must, v11+ is needed. Any already existing Postgres database will do the trick, see the *Bootstrapping the Metrics DB* section for2 details.

- TimescaleDB time-series extension for PostgreSQL

  Although technically a plain extension it's often mentioned as a separate database system as it brings custom data compression to the table, enabling huge disk savings over standard Postgres. Note that pgwatch2 does not use Timescale's built-in *retention* management but a custom version.

- Prometheus Time Series DB and monitoring system

  Though Prometheus is not a traditional database system, it's a good choice for monitoring Cloud-like environments as the monitoring targets don't need to know too much about how actual monitoring will be carried out later and also Prometheus has a nice fault-tolerant alerting system for enterprise needs. NB! By default Prometheus is not set up for long term metrics storage!

- Graphite Time Series DB

  Not as modern as the other options but a performant TSDB nevertheless with built-in charting. In pgwatch2 use case though there's no support for "custom tags" and request batching support which should not be a big problem for lighter use cases.

- JSON files

  Plain text files for testing / special use cases.

5.4 The Web UI

The second homebrewn component of the pgwatch2 solution is an optional and relatively simple Web UI for administering details of the monitoring configuration like which databases should be monitored, with which metrics and intervals. Besides that there are some basic overview tables to analyze the gathered data and also possibilities to delete unneeded metric data (when removing a test host for example).

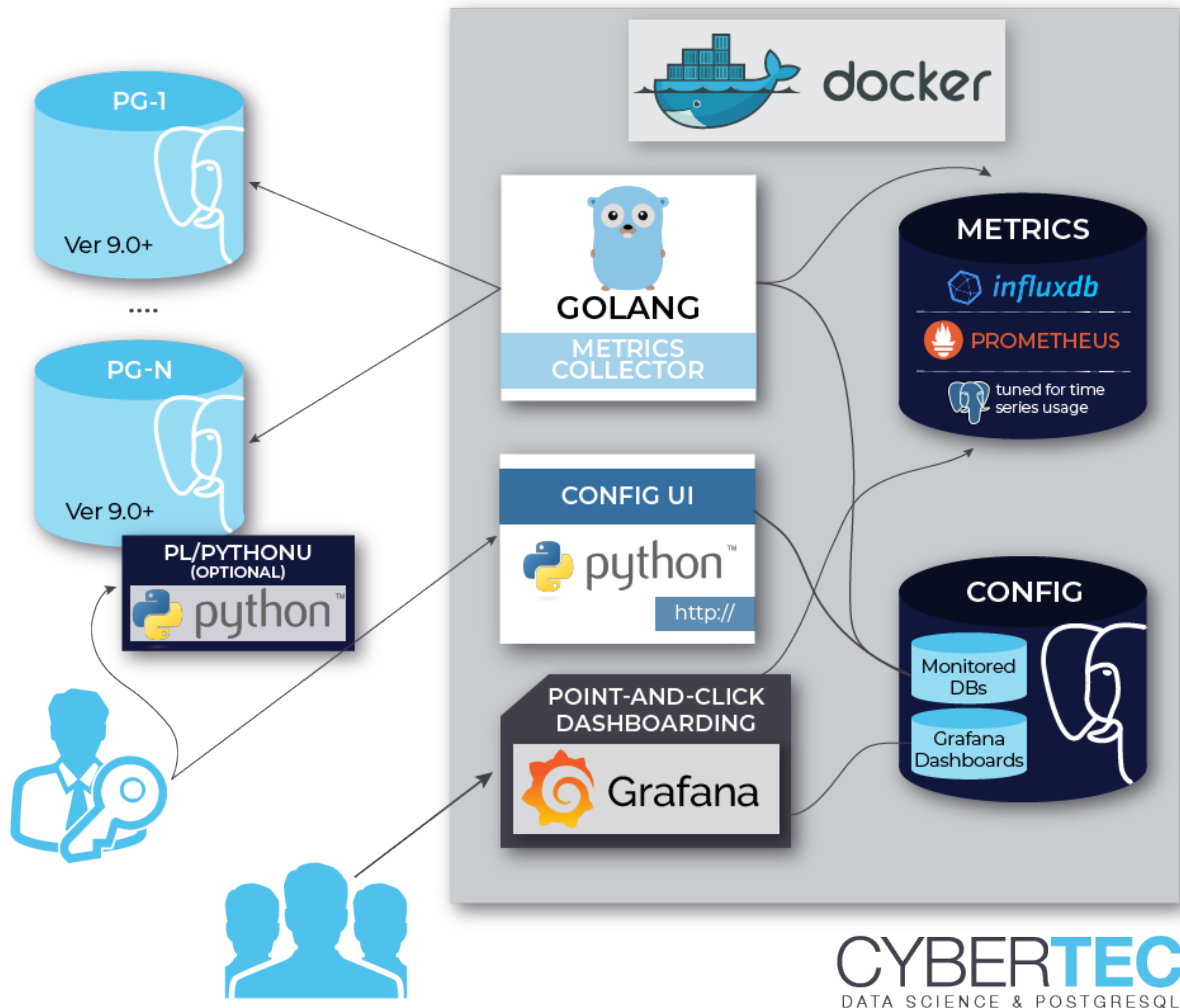
NB! Note that the Web UI can only be used if storing the configuration in the database (Postgres).

5.5 Metrics representation

Standard pgwatch2 setup uses Grafana for analyzing the gathered metrics data in a visual, point-and-click way. For that a rich set of predefined dashboards for Postgres and InfluxDB data sources is provided, that should cover the needs of most users - advanced users would mostly always want to customize some aspects though, so it's not meant as a one-size-fits-all solution. Also as metrics are stored in a DB, they can be visualized or processed in any other way.

5.6 Component diagram

Component diagram of a typical setup:



5.7 Component reuse

NB! All components are *loosely coupled*, thus for non-pgwatch2 components (pgwatch2 components are only the metrics collector and the optional Web UI) you can decide to make use of an already existing installation of Postgres, Grafana or InfluxDB and run additionally just the pgwatch2 collector.

- To use an existing Postgres DB for storing the monitoring config

  Create a new pgwatch2 DB, preferrably also an accroding role who owns it. Then roll out the schema (pgwatch2/sql/config_store/config_store.sql) and set the following parameters when running the image: PW2_PGHOST, PW2_PGPORT, PW2_PGDATABASE, PW2_PGUSER, PW2_PGPASSWORD, PW2_PGSSL (optional).

- To use an existing Grafana installation

Load the pgwatch2 dashboards from *grafana_dashboard* folder if needed (one can totally define their own) and set the following paramater: PW2_GRAFANA_BASEURL. This parameter only provides correct links to Grafana dashboards from the Web UI. Grafana is the most loosely coupled component for pgwatch2 and basically doesn't have to be used at all. One can make use of the gathered metrics directly over the Influx (or Graphite) API-s.

- To use an existing InfluxDB installation

  Set the following env variables: PW2_IHOST, PW2_IPORT, PW2_IDATABASE, PW2_IUSER, PW2_IPASSWORD, PW2_ISSL (optional).

  NB! Note that if wanting to use SSL with self-signed certificates on InfluxDB side then some extra steps described in this Github issue are needed.

- To use an existing Graphite installation

  One can also store the metrics in Graphite instead of InfluxDB (no predefined pgwatch2 dashboards for Graphite though). Following parameters needs to be set then: PW2_DATASTORE=graphite, PW2_GRAPHITEHOST, PW2_GRAPHITEPORT

- To use an existing Postgres DB for storing metrics

  1. Roll out the metrics storage schema according to instructions from *here*.
  2. Following parameters need to be set for the gatherer:

  - `--datastore=postgres` or `PW2_DATASTORE=postgres`
  - `--pg-metric-store-conn-str="postgresql://user:pwd@host:port/db"` or `PW2_PG_METRIC_STORE_CONN_STR="..."`
  - optionally also adjust the `--pg-retention-days` parameter. By default 30 days for InfluxDB and 14 days for Postgres are kept

  3. If using the Web UI also set the datastore parameters `--datastore` and `--pg-metric-store-conn-str` if wanting to have an option to be able to clean up data also via the UI in a more targeted way.

  NB! When using Postgres metrics storage, the schema rollout script activates "asynchronous commiting" feature for the *pgwatch2* role in the metrics storage DB by default! If this is not wanted (no metrics can be lost in case of a crash), then re-enstate normal (synchronous) commiting with below query and restart the pgwatch2 agent:

```
ALTER ROLE pgwatch2 IN DATABASE $MY_METRICS_DB SET synchronous_commit TO on;
```

CHAPTER 6

Installation options

Besides freedom of choosing from a set of metric storage options one can also choose how they're going to retrieve metrics from databases - in a "pull" or "push" way and how is the monitoring configuration (connect strings, metric sets and intervals) going to be stored.

6.1 Config DB based operation

This is the original central pull mode depicted on the *architecture diagram*. It requires a small schema to be rolled out on any Postgres database accessible to the metrics gathering daemon, which will hold the connect strings, metric definition SQL-s and preset configurations and some other more minor attributes. For rollout details see the *custom installation* chapter.

The default Docker images use this approach.

6.2 File based operation

From v1.4.0 one can deploy the gatherer daemon(s) decentrally with *hosts to be monitored* defined in simple YAML files. In that case there is no need for the central Postgres "config DB". See the sample instances.yaml config file for an example. Note that in this mode you also need to point out the path to metric definition SQL files when starting the gatherer. Also note that the configuration system also supports multiple YAML files in a folder so that you could easily programmatically manage things via *Ansible* for example and you can also use Env. vars in sideYAML files.

Relevant Gatherer env. vars / flags: `--config`, `--metrics-folder` or `PW2_CONFIG` / `PW2_METRICS_FOLDER`.

6.3 Ad-hoc mode

Optimized for Cloud scenarios and quick temporary setups, it's also possible to run the metrics gathering daemon in a somewhat limited "ad-hoc" mode, by specifying a single connection string via ENV or command line input, plus

the same for the metrics and intervals (as a JSON string) or a preset config name. In this mode it's only possible to monitor a single specified Postgres DB (the default behaviour) or the whole instance, i.e. all non-template databases found on the instance.

This mode is perfect also for Cloud setups in *sidecar* constellations when exposing the Prometheus output. Pushing to a central metrics DB is also a good option for non-cloud setups if monitoring details like intervals etc are static enough - it helps to distribute the metrics gathering load and is more fault-tolerant. The only critical component will then be the metrics storage DB, but that can be well solved with a HA solution like Patroni for example.

Main benefit - in this mode there is no need for the central Postgres "config DB" nor any YAML config files. NB! When using that mode with the default Docker image, the built-in metric definitions can't be changed via the Web UI and it's actually recommended to use the *gatherer only* image named *cybertec/pgwatch2-daemon*.

Relevant Gatherer env. vars / flags: `--adhoc-conn-str, --adhoc-config, --adhoc-name, --metrics-folder` or respectively `PW2_ADHOC_CONN_STR, PW2_ADHOC_CONFIG, PW2_ADHOC_NAME, PW2_METRICS_FOLDER, PW2_ADHOC_CREATE_HELPERS`.

```
# launching in ad-hoc / test mode
docker run --rm -p 3000:3000 -e PW2_ADHOC_CONN_STR="postgresql://user:pwd@mydb:5432/
↪mydb1" \
    -e PW2_ADHOC_CONFIG=unprivileged --name pw2 cybertec/pgwatch2-postgres

# launching in ad-hoc / test mode, creating metrics helpers automatically (requires␣
↪superuser)
docker run --rm -p 3000:3000 -e PW2_ADHOC_CONN_STR="postgresql://user:pwd@mydb:5432/
↪mydb1" \
    -e PW2_ADHOC_CONFIG=exhaustive -e PW2_ADHOC_CREATE_HELPERS=true --name pw2␣
↪cybertec/pgwatch2-postgres
```

NB! Using the `PW2_ADHOC_CREATE_HELPERS` flag will try to create all metrics fetching helpers automatically if not already existing - this assumes superuser privileges though, which is not recommended for long term setups for obvious reasons. In case a long term need rises it's recommended to change the monitoring role to an unprivileged *pgwatch2* user, which by default gets execute *GRANT*-s to all helper functions. More details on how to deal with *helpers* can be found *here* and more on secure setups in the *security chapter*.

6.4 Prometheus mode

In v1.6.0 was added support for Prometheus - being one of the most popular modern metrics gathering / alerting solutions. When the `--datastore / PW2_DATASTORE` parameter is set to *prometheus* then the pgwatch2 metrics collector doesn't do any normal interval-based fetching but listens on port *9187* (changeable) for scrape requests configured and performed on Prometheus side. Returned metrics belong to the "pgwatch2" namespace (a prefix basically) which is changeable via the `--prometheus-namespace` flag if needed.

Also important to note - in this mode the pgwatch2 agent should not be run centrally but on all individual DB hosts. While technically possible though to run centrally, it would counter the core idea of Prometheus and would make scrapes also longer and risk getting timeouts as all DBs are scanned sequentially for metrics.

FYI – the functionality has some overlap with the existing postgres_exporter project, but also provides more flexibility in metrics configuration and all config changes are applied "online".

Also note that Prometheus can only store numerical metric data values - so metrics output for example PostgreSQL storage and Prometheus are not 100% compatile. Due to that there's also a separate "preset config" named "prometheus".

CHAPTER 7

Installing using Docker

7.1 Simple setup steps

The simplest real-life pgwatch2 setup should look something like that:

1. Decide which metrics storage engine you want to use - *cybertec/pgwatch2* and *cybertec/pgwatch2-nonroot* images use InfluxDB internally for metrics storage, while *cybertec/pgwatch2-postgres* uses PostgreSQL. For Prometheus mode (exposing a port for remote scraping) one should use the slimmer *cybertec/pgwatch2-daemon* image which doesn't have any built in databases.
2. Find the latest pgwatch2 release version by going to the project's Github *Releases* page or use the public API with something like that:

   ```
   curl -so- https://api.github.com/repos/cybertec-postgresql/pgwatch2/releases/
   ↪latest | jq .tag_name | grep -oE '[0-9\.]+'
   ```

3. Pull the image:

   ```
   docker pull cybertec/pgwatch2:X.Y.Z
   ```

4. Run the Docker image, exposing minimally the Grafana port served on port 3000 internally. In a relatively secure environment you'd usually also include the administrative web UI served on port 8080:

   ```
   docker run -d --restart=unless-stopped -p 3000:3000 -p 8080:8080 --name pw2␣
   ↪cybertec/pgwatch2:X.Y.Z
   ```

   Note that we're using a Docker image with the built-in InfluxDB metrics storage DB here and setting the container to be automatically restarted in case of a reboot / crash - which is highly recommended if not using some container management framework to run pgwatch2.

7.2 More future proof setup steps

Although the above simple setup example will do for more temporal setups / troubleshooting sessions, for permanent setups it's highly recommended to create separate volumes for all software components in the container, so that it would be easier to *update* to newer pgwatch2 Docker images and pull file system based backups and also it might be a good idea to expose all internal ports at least on *localhost* for possible troubleshooting and making possible to use native backup tools more conveniently for InfluxDB or Postgres.

Note that for maximum flexibility, security and update simplicity it's best to do a custom setup though - see the next *chapter* for that.

So in short, for plain Docker setups would be best to do something like:

```
# let's create volumes for Postgres, Grafana and pgwatch2 marker files / SSL␣
↪certificates
for v in pg  grafana pw2 ; do docker volume create $v ; done

# launch pgwatch2 with fully exposed Grafana and Health-check ports
# and local Postgres and subnet level Web UI ports
docker run -d --restart=unless-stopped --name pw2 \
  -p 3000:3000 -p 8081:8081 -p 127.0.0.1:5432:5432 -p 192.168.1.XYZ:8080:8080 \
  -v pg:/var/lib/postgresql -v grafana:/var/lib/grafana -v pw2:/pgwatch2/persistent-
↪config \
  cybertec/pgwatch2-postgres:X.Y.Z
```

Note that in non-trusted environments it's a good idea to specify more sensitive ports together with some explicit network interfaces for additional security - by default Docker listens on all network devices!

Also note that one can configure many aspects of the software components running inside the container via ENV - for a complete list of all supported Docker environment variables see the ENV_VARIABLES.md file.

7.3 Available Docker images

Following images are regularly pushed to Docked Hub:

cybertec/pgwatch2 The original pgwatch2 "batteries-included" image with InfluxDB metrics storage. Just insert connect infos to your database via the admin Web UI (or directly into the Config DB) and then turn to the pre-defined Grafana dashboards to analyze DB health and performance.

cybertec/pgwatch2-postgres Exactly the same as previous, but metrics are also stored in PostgreSQL - thus needs more disk space. But in return you get more "out of the box" dashboards, as the power of standard SQL gives more complex visualization options.

cybertec/pgwatch2-nonroot Same components as for the original *cybertec/pgwatch2* image, but no "root" user is used internally, so it can also be launched in security restricted environments like OpenShift. Limits ad-hoc troubleshooting and "in container" customizations or updates though, but this is the standard for orchestrated cloud environments - you need to fix the image and re-deploy.

cybertec/pgwatch2-daemon A light-weight image containing only the metrics collection daemon / agent, that can be integrated into the monitoring setup over configuration specified either via ENV, mounted YAML files or a PostgreSQL Config DB. See the *Component reuse* chapter for wiring details.

cybertec/pgwatch2-db-bootstrapper Sole purpose of the image is to bootstrap the pgwatch2 *Config DB* or *Metrics DB* schema. Useful for custom cloud oriented setups where the above "all components included" images are not a good fit.

7.4 Building custom Docker images

For custom tweaks, more security,specific component versions, etc one could easily build the images themselves, just a Docker installation is needed: *docker build .*.

Build scripts used to prepare the public images can be found here.

7.5 Interacting with the Docker container

- If to launch with the *PW2_TESTDB=1* env. parameter then the pgwatch2 configuration database running inside Docker is added to the monitoring, so that you should immediately see some metrics at least on the *Health-check* dashboard.
- To add new databases / instances to monitoring open the administration Web interface on port 8080 (or some other port, if re-mapped at launch) and go to the */dbs* page. Note that the Web UI is an optional component, and one can managed monitoring entries directly in the Postgres Config DB via INSERT-s / UPDATE-s into "pgwatch2.monitored_db" table. Default user/password are again *pgwatch2* / *pgwatch2admin*, database name - pgwatch2. In both cases note that it can take up to 2min (default main loop time, changeable via *PW2_SERVERS_REFRESH_LOOP_SECONDS*) before you see any metrics for newly inserted databases.
- One can edit existing or create new Grafana dashboards, change Grafana global settings, create users, alerts, etc after logging in as *admin* / *pgwatch2admin* (by default, changeable at launch time).
- Metrics and their intervals that are to be gathered can be customized for every database separately via a custom JSON config field or more conveniently by using *Preset Configs*, like "minimal", "basic" or "exhaustive" (monitored_db.preset_config table), where the name should already hint at the amount of metrics gathered. For privileged users the "exhaustive" preset is a good starting point, and "unprivileged" for simple developer accounts.
- To add a new metrics yourself (which are simple SQL queries returning any values and a timestamp) head to http://127.0.0.1:8080/metrics. The queries should always include a "epoch_ns" column and "tag_" prefix can be used for columns that should be quickly searchable / groupable, and thus will be indexed with the InfluxDB and PostgreSQL metric stores. See to the bottom of the "metrics" page for more explanations or the documentation chapter on metrics *here*.
- For a quickstart on dashboarding, a list of available metrics together with some instructions are presented on the "Documentation" dashboard.
- Some built-in metrics like "cpu_load" and others, that gather privileged or OS statistics, require installing *helper functions* (looking like that, so it might be normal to see some blank panels or fetching errors in the logs. On how to prepare databases for monitoring see the *Monitoring preparations* chapter.
- For effective graphing you want to familiarize yourself with the query language of the database system that was selected for metrics storage. Some tips to get going:
  - For InfluxQL - the non_negative_derivative() function is very handy as Postgres statistics are mostly ever-growing counters and one needs to calculate so called *deltas* to show change. Documentation here.
  - For PostgreSQL / TimescaleDB - some knowledge of Window functions is a must if looking at longer time periods of data as the statistics could have been reset in the mean time by user request or the server might have crashed, so that simple *max() - min()* aggregates on cumulative counters (most data provided by Postgres is cumulative) would lie.
- For possible troubleshooting needs, logs of the components running inside Docker are by default (if not disabled on container launch) visible under: http://127.0.0.1:8080/logs/[pgwatch2|postgres|webui|influxdb|grafana]. It's of course also possible to log into the container and look at log files directly - they're situated under */var/logs/supervisor/*.

FYI - `docker logs ...` command is not really useful after a successful container startup in pgwatch2 case.

7.6 Ports used

- 5432 - Postgres configuration or metrics storage DB (when using the cybertec/pgwatch2-postgres image)
- 8080 - Management Web UI (monitored hosts, metrics, metrics configurations)
- 8081 - Gatherer healthcheck / statistics on number of gathered metrics (JSON).
- 3000 - Grafana dashboarding
- 8086 - InfluxDB API (when using the InfluxDB version)
- 8088 - InfluxDB Backup port (when using the InfluxDB version)

7.7 Docker Compose

As mentioned in the *Components* chapter, remember that the pre-built Docker images are just one example how your monitoring setup around the pgwatch2 metrics collector could be organized. For another example how various components (as Docker images here) can work together, see a *Docker Compose* example with loosely coupled components here.

CHAPTER 8

Custom installation

As described in the *Components* chapter, there a couple of ways how to set up up pgwatch2. Two most common ways though are the central *Config DB* based "pull" approach and the *YAML file* based "push" approach, plus Grafana to visualize the gathered metrics.

8.1 Config DB based setup

Overview of installation steps

1. Install Postgres or use any available existing instance - v9.4+ required for the config DB and v11+ for the metrics DB.
2. Bootstrap the Config DB.
3. Bootstrap the metrics storage DB (PostgreSQL here).
4. Install pgwatch2 - either from pre-built packages or by compiling the Go code.
5. Prepare the "to-be-monitored" databases for monitoring by creating a dedicated login role name as a minimum.
6. Optional step - install the administrative Web UI + Python & library dependencies.
7. Add some databases to the monitoring configuration via the Web UI or directly in the Config DB.
8. Start the pgwatch2 metrics collection agent and monitor the logs for any problems.
9. Install and configure Grafana and import the pgwatch2 sample dashboards to start analyzing the metrics.
10. Make sure that there are auto-start SystemD services for all components in place and optionally set up also backups.

Detailed steps for the Config DB based "pull" approach with Postgres metrics storage

Below are sample steps to do a custom install from scratch using Postgres for the pgwatch2 configuration DB, metrics DB and Grafana config DB.

All examples here assume Ubuntu as OS - but it's basically the same for RedHat family of operations systems also, minus package installation syntax differences.

1. **Install Postgres**

   Follow the standard Postgres install procedure basically. Use the latest major version available, but minimally v11+ is recommended for the metrics DB due to recent partitioning speedup improvements and also older versions were missing some default JSONB casts so that a few built-in Grafana dashboards need adjusting otherwise.

   To get the latest Postgres versions, official Postgres PGDG repos are to be preferred over default disto repos. Follow the instructions from:

   - https://wiki.postgresql.org/wiki/Apt - for Debian / Ubuntu based systems
   - https://www.postgresql.org/download/linux/redhat/ - for CentOS / RedHat based systems

2. **Install pgwatch2** - either from pre-built packages or by compiling the Go code

   - Using pre-built packages

     The pre-built DEB / RPM / Tar packages are available on the Github releases page.

     ```
     # find out the latest package link and replace below, using v1.8.0 here
     wget https://github.com/cybertec-postgresql/pgwatch2/releases/download/v1.8.0/
     ↪pgwatch2_v1.8.0-SNAPSHOT-064fdaf_linux_64-bit.deb
     sudo dpkg -i pgwatch2_v1.8.0-SNAPSHOT-064fdaf_linux_64-bit.deb
     ```

   - Compiling the Go code yourself

     This method of course is not needed unless dealing with maximum security environments or some slight code changes are required.

     1. Install Go by following the official instructions
     2. Get the pgwatch2 project's code and compile the gatherer daemon

        ```
        git clone https://github.com/cybertec-postgresql/pgwatch2.git
        cd pgwatch2/pgwatch2
        ./build_gatherer.sh
        # after fetching all the Go library dependencies (can take minutes)
        # an executable named "pgwatch2" should be generated. Additionally it's a␣
        ↪good idea
        # to copy it to /usr/bin/pgwatch2-daemon as that's what the default␣
        ↪SystemD service expects.
        ```

   - Configure a SystemD auto-start service (optional)

     Sample startup scripts can be found at */etc/pgwatch2/startup-scripts/pgwatch2.service* or online here. Note that they are OS agnostic and might need some light adjustment of paths, etc - so always test them out.

3. **Boostrap the config DB**

   1. Create a user to "own" the pgwatch2 schema

      Typically called *pgwatch2* but can be anything really, if the schema creation file is adjusted accordingly.

      ```
      psql -c "create user pgwatch2 password 'xyz'"
      psql -c "create database pgwatch2 owner pgwatch2"
      ```

   2. Roll out the pgwatch2 config schema

      The schema will most importantly hold connection strings of DB-s to be monitored and the metric definitions.

```
# FYI - one could get the below schema files also directly from Github
# if re-using some existing remote Postgres instance where pgwatch2 was not␣
↪installed
psql -f /etc/pgwatch2/sql/config_store/config_store.sql pgwatch2
psql -f /etc/pgwatch2/sql/config_store/metric_definitions.sql pgwatch2
```

1. **Bootstrap the metrics storage DB**

   1. Create a dedicated database for storing metrics and a user to "own" the metrics schema

      Here again default scripts expect a role named "pgwatch2" but can be anything if to adjust the scripts.

      ```
      psql -c "create database pgwatch2_metrics owner pgwatch2"
      ```

   2. Roll out the pgwatch2 metrics storage schema

      This is a place to pause and first think how many databases will be monitored, i.e. how much data generated, and based on that one should choose an according metrics storage schema. There are a couple of different options available that are described here in detail, but the gist of it is that you don't want too complex partitioning schemes if you don't have zounds of data and don't need the fastest queries. For a smaller amount of monitored DBs (a couple dozen to a hundred) the default "metric-time" is a good choice. For hundreds of databases, aggressive intervals, or long term storage usage of the TimescaleDB extension is recommended.

      ```
      cd /etc/pgwatch2/sql/metric_store
      psql -f roll_out_metric_time.psql pgwatch2_metrics
      ```

      NB! Default retention for Postgres storage is 2 weeks! To change, use the `--pg-retention-days / PW2_PG_RETENTION_DAYS` gatherer parameter.

2. **Prepare the "to-be-monitored" databases for metrics collection**

   As a minimum we need a plain unprivileged login user. Better though is to grant the user also the *pg_monitor* system role, available on v10+. Superuser privileges should be normally avoided for obvious reasons of course, but for initial testing in safe environments it can make the initial preparation (automatic *helper* rollouts) a bit easier still, given superuser privileges are later stripped.

   NB! To get most out of your metrics some *SECURITY DEFINER* wrappers functions called "helpers" are recommended on the DB-s under monitoring. See the detailed chapter on the "preparation" topic *here* for more details.

3. **Install Python 3 and start the Web UI (optional)**

   NB! The Web UI is not strictly required but makes life a lot easier for *Config DB* based setups. Technically it would be fine also to manage connection strings of the monitored DB-s directly in the "pgwatch2.monitored_db" table and add/adjust metrics in the "pgwatch2.metric" table, and *Preset Configs* in the "pgwatch2.preset_config" table.

   1. Install Python 3 and Web UI requirements

      ```
      # first we need Python 3 and "pip" - the Python package manager
      sudo apt install python3 python3-pip
      cd /etc/pgwatch2/webpy/
      sudo pip3 install -U -r requirements_pg_metrics.txt
      # NB! Replace with "requirements_influx_metrics.txt" if using InfluxDB to␣
      ↪store metrics
      ```

   2. Exposing component logs (optional)

For use cases where exposing the component (Grafana, Postgres, Influx, gatherer daemon, Web UI itself) logs over the "/logs" endpoint remotely is wanted, then in the custom setup mode some actual code changes are needed to specify where logs of all components are situated - see top of the pgwatch2.py file for that. Defaults are set to work with the Docker image.

3. Start the Web UI

```
# NB! The defaults assume a local Config DB named pgwatch2, DB user pgwatch2
python3 web.py --datastore=postgres --pg-metric-store-conn-str=
↪"dbname=pgwatch2_metrics user=pgwatch2"
```

Default port for the Web UI: **8080**. See `web.py --help` for all options.

4. Configure a SystemD auto-start service (optional)

Sample startup scripts can be found at */etc/pgwatch2/webpy/startup-scripts/pgwatch2-webui.service* or online here. Note that they are OS agnostic and always need some light adjustment of paths, etc - so always test them out.

4. **Configure DB-s and metrics / intervals to be monitored**
   - From the Web UI "/dbs" page
   - Via direct inserts into the Config DB *pgwatch2.monitored_db* table

5. **Start the pgwatch2 metrics collection agent**

1. The gatherer has quite some parameters (use the *–help* flag to show them all), but simplest form would be:

```
# default connections params expect a trusted localhost Config DB setup
# so mostly the 2nd line is not needed actually
pgwatch2-daemon \
  --host=localhost --user=pgwatch2 --dbname=pgwatch2 \
  --datastore=postgres --pg-metric-store-conn-str=postgresql://
↪pgwatch2@localhost:5432/pgwatch2_metrics \
  --verbose=info

# or via SystemD if set up in step #2
useradd -m -s /bin/bash pgwatch2 # default SystemD templates run under the␣
↪pgwatch2 user
sudo systemctl start pgwatch2
sudo systemctl status pgwatch2
```

After initial verification that all works it's usually good idea to set verbosity back to default by removing the *verbose* flag.

Another tip to configure connection strings inside SystemD service files is to use the "systemd-escape" utility to escape special characters like spaces etc if using the LibPQ connect string syntax rather than JDBC syntax.

2. Alternative start command when using InfluxDB storage:

```
pgwatch2-daemon \
  --host=localhost --user=pgwatch2 --dbname=pgwatch2 \
  --datastore=influx \
  --ihost=my-influx-db --idbname=pgwatch2 --iuser=pgwatch2 --ipassword=xyz
```

NB! pgwatch2 has also support for writing metrics into two separate Influx databases in parallel as the Open Source version has no HA options comparable to Postgres.

3. Monitor the console or log output for any problems

If you see metrics trickling into the "pgwatch2_metrics" database (metric names are mapped to table names and tables are auto-created), then congratulations - the deployment is working! When using some more aggressive *preset metrics config* then there are usually still some errors though, due to the fact that some more extensions or privileges are missing on the monitored database side. See the according chapter *here*.

NB! When you're compiling your own gatherer then the executable file will be named just *pgwatch2* instead of *pgwatch2-daemon* to avoid mixups.

1. **Install Grafana**

   1. Create a Postgres database to hold Grafana internal config, like dashboards etc

      Theoretically it's not absolutely required to use Postgres for storing Grafana internal settings / dashboards, but doing so has 2 advantages - you can easily roll out all pgwatch2 built-in dashboards and one can also do remote backups of the Grafana configuration easily.

      ```
      psql -c "create user pgwatch2_grafana password 'xyz'"
      psql -c "create database pgwatch2_grafana owner pgwatch2_grafana"
      ```

   2. Follow the instructions from https://grafana.com/docs/grafana/latest/installation/debian/, basically something like:

      ```
      wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -
      echo "deb https://packages.grafana.com/oss/deb stable main" | sudo tee -a /
      ↪etc/apt/sources.list.d/grafana.list
      sudo apt-get update && sudo apt-get install grafana

      # review / change config settings and security, etc
      sudo vi /etc/grafana/grafana.ini

      # start and enable auto-start on boot
      sudo systemctl daemon-reload
      sudo systemctl start grafana-server
      sudo systemctl status grafana-server
      ```

      Default Grafana port: 3000

   3. Configure Grafana config to use our pgwatch2_grafana DB

      Place something like below in the "[database]" section of /etc/grafana/grafana.ini

      ```
      [database]
      type = postgres
      host = my-postgres-db:5432
      name = pgwatch2_grafana
      user = pgwatch2_grafana
      password = xyz
      ```

      Taking a look at [server], [security] and [auth*] sections is also recommended.

   4. Set up the pgwatch2 metrics database as the default datasource

      We need to tell Grafana where our metrics data is located. Add a datasource via the Grafana UI (Admin -> Data sources) or adjust and execute the "pgwatch2/bootstrap/grafana_datasource.sql" script on the *pgwatch2_grafana* DB.

   5. Add pgwatch2 predefined dashboards to Grafana

      This could be done by importing the pgwatch2 dashboard definition JSON-s manually, one by one, from the "grafana_dashboards" folder ("Import Dashboard" from the Grafana top menu) or via as small helper

script located at */etc/pgwatch2/grafana-dashboards/import_all.sh*. The script needs some adjustment for metrics storage type, connect data and file paths.

6. Optionally install also Grafana plugins

   Currently one pre-configured dashboard (Biggest relations treemap) use an extra plugin - if planning to that dash, then run the following:

   ```
   grafana-cli plugins install savantly-heatmap-panel
   ```

7. Start discovering the preset dashbaords

   If the previous step of launching pgwatch2 daemon succeeded and it was more than some minutes ago, one should already see some graphs on dashboards like "DB overview" or "DB overview Unprivileged / Developer mode" for example.

8.2 YAML based setup

From v1.4 one can also deploy the pgwatch2 gatherer daemons more easily in a de-centralized way, by specifying monitoring configuration via YAML files. In that case there is no need for a central Postgres "config DB".

YAML installation steps

1. Install pgwatch2 - either from pre-built packages or by compiling the Go code.
2. Specify hosts you want to monitor and with which metrics / aggressivness in a YAML file or files, following the example config located at */etc/pgwatch2/config/instances.yaml* or online here. Note that you can also use env. variables inside the YAML templates!
3. Bootstrap the metrics storage DB (not needed it using Prometheus mode).
4. Prepare the "to-be-monitored" databases for monitoring by creating a dedicated login role name as a *minimum*.
5. Run the pgatch2 gatherer specifying the YAML config file (or folder), and also the folder where metric definitions are located. Default location: */etc/pgwatch2/metrics*.
6. Install and configure Grafana and import the pgwatch2 sample dashboards to start analyzing the metrics. See above for instructions.
7. Make sure that there are auto-start SystemD services for all components in place and optionally set up also backups.

Relevant gatherer parameters / env. vars: `--config` / `PW2_CONFIG` and `--metrics-folder` / `PW2_METRICS_FOLDER`.

For details on individual steps like installing pgwatch2 see the above paragraph.

NB! The Web UI component cannot be used in file based mode.

8.3 Using InfluxDB for metrics storage

An alternative flow for the above examples would be to replace Postgres metrics storage with InfluxDB. This might be a good idea when you have hundreds of databases to monitor or want to use very aggressive intervals as InfluxDB has the smallest disk footprint of the supported options (with more CPU / RAM usage though). See the *Sizing recommendations* chapter for indicative numbers.

1. Install InfluxDB (the Open Source version)

   1. From project package repositories:

Follow the instructions from https://docs.influxdata.com/influxdb/latest/introduction/install/ or just download and install the latest package:

1. Or directly from the packages:

```
INFLUX_LATEST=$(curl -so- https://api.github.com/repos/influxdata/influxdb/
↪releases/latest \
                  | jq .tag_name | grep -oE '[0-9\.]+')
wget https://dl.influxdata.com/influxdb/releases/influxdb_${INFLUX_LATEST}_
↪amd64.deb
sudo dpkg -i influxdb_${INFLUX_LATEST}_amd64.deb
```

2. Review / adjust the config and start the server

   Take a look at the default config located at */etc/influxdb/influxdb.conf* and edit per use case / hardware needs. Most importantly one should enable authentication if not running InfluxDB on the same host as the collector or to set the server to listen only on localhost (the *bind-address* parameter).

   Also changing the *wal-fsync-delay* parameter usually makes sense to get better performance, as metric data is usually something where we can in the worst case lose the latest half a second of data without problems.

   See here for more information on configuring InfluxDB.

3. Create a non-root user, a metrics database and a retention policy (optional)

   If security is topic one should create a separate non-root login user (e.g. "pgwatch2") to be used by the metrics gathering daemon to store metrics. See here for details on creating new users.

   If going that road one also needs to create manually a database and a retention policy to go with it as by default old metrics data is not purged. These tasks by the way are also tried by the pgwatch2 daemon automatically, but will fail if not an admin user.

   Sample commands:

```
CREATE DATABASE pgwatch2 WITH DURATION 30d REPLICATION 1 SHARD DURATION 1d NAME␣
↪pgwatch2_def_ret
CREATE USER pgwatch2 WITH PASSWORD 'qwerty'
GRANT READ ON pgwatch2 TO pgwatch2
GRANT WRITE ON pgwatch2 TO pgwatch2
```

Default port for the InfluxDB client API: **8086**

CHAPTER 9

Preparing databases for monitoring

9.1 Effects of monitoring

- Although the “Observer effect” applies also for pgwatch2, no noticeable impact for the monitored DB is expected when using *Preset configs* settings, and given that there is some normal load on the server anyways and the DB doesn’t have thousands of tables. For some metrics though can happen that the metric reading query (notably “stat_statements” and “table_stats”) takes some tens of milliseconds, which might be more than an average application query.
- At any time maximally 2 metric fetching queries can run in parallel on any monitored DBs. This can be changed by recompiling (MAX_PG_CONNECTIONS_PER_MONITORED_DB variable) the gatherer.
- Default Postgres statement timeout is *5s* for entries inserted via the Web UI / database directly.

9.2 Basic preparations

As a base requirement you’ll need a **login user** (non-superuser suggested) for connecting to your server and fetching metrics.

Though theoretically you can use any username you like, but if not using “pgwatch2” you need to adjust the “helper” creation SQL scripts (see below for explanation) accordingly, as in those by default the “pgwatch2” will be granted execute privileges.

```
CREATE ROLE pgwatch2 WITH LOGIN PASSWORD 'secret';
-- NB! For critical databases it might make sense to ensure that the user account
-- used for monitoring can only open a limited number of connections
-- (there are according checks in code, but multiple instances might be launched)
ALTER ROLE pgwatch2 CONNECTION LIMIT 3;
GRANT pg_monitor TO pgwatch2;   // v10+
GRANT CONNECT ON DATABASE mydb TO pgwatch2;
```

(continues on next page)

(continued from previous page)

```
GRANT USAGE ON SCHEMA public TO pgwatch2; -- NB! pgwatch doesn't necessarily require␣
↪using the public schema though!
GRANT EXECUTE ON FUNCTION pg_stat_file(text) to pgwatch2; -- needed by the wal_size␣
↪metric
```

For most monitored databases it's extremely beneficial (to troubleshooting performance issues) to also activate the pg_stat_statements extension which will give us exact "per query" performance aggregates and also enables to calculate how many queries are executed per second for example. In pgwatch2 context it powers the "Stat statements Top" dashboard and many other panels of other dashboards. For additional troubleshooting benefits also the track_io_timing setting should be enabled.

1. Make sure the Postgres *contrib* package is installed (should be installed automatically together with the Postgres server package on Debian based systems).
   - On RedHat / Centos: `yum install -y postgresqlXY-contrib`
   - On Debian / Ubuntu: `apt install postgresql-contrib`
2. Add *pg_stat_statements* to your server config (postgresql.conf) and restart the server.

```
shared_preload_libraries = 'pg_stat_statements'
track_io_timing = on
```

3. After restarting activate the extension in the monitored DB. Assumes Postgres superuser.

```
psql -c "CREATE EXTENSION IF NOT EXISTS pg_stat_statements"
```

9.3 Rolling out helper functions

Helper functions in pgwatch2 context are standard Postgres stored procedures, running under *SECURITY DEFINER* privileges. Via such wrapper functions one can do **controlled privilege escalation** - i.e. to give access to protected Postgres metrics (like active session details, "per query" statistics) or even OS-level metrics, to normal unprivileged users, like the pgwatch2 monitoring role.

If using a superuser login (recommended only for local "push" setups) you have full access to all Postgres metrics and would need *helpers* only for OS remote statistics. For local (push) setups as of pgwatch2 version 1.8.4 the most typical OS metrics are covered by the "–direct-os-stats" flag, explained below.

For unprivileged monitoring users it is highly recommended to take these additional steps on the "to be monitored" database to get maximum value out of pgwatch2 in the long run. Without these additional steps, you lose though about 10-15% of built-in metrics, which might not be too tragical nevertheless. For that use case there's also a *preset config* named "unprivileged".

NB! When monitoring v10+ servers then the built-in **pg_monitor** system role is recommended for the monitoring user, which almost substitutes superuser privileges for monitoring purposes in a safe way.

Rolling out common helpers

For completely unprivileged monitoring users the following *helpers* are recommended to make good use of the default "exhaustive" *Preset Config*:

```
export PGUSER=superuser
psql -f /etc/pgwatch2/metrics/00_helpers/get_stat_activity/$pgver/metric.sql mydb
psql -f /etc/pgwatch2/metrics/00_helpers/get_stat_replication/$pgver/metric.sql mydb
psql -f /etc/pgwatch2/metrics/00_helpers/get_wal_size/$pgver/metric.sql mydb
```

(continues on next page)

(continued from previous page)

```
psql -f /etc/pgwatch2/metrics/00_helpers/get_stat_statements/$pgver/metric.sql mydb
psql -f /etc/pgwatch2/metrics/00_helpers/get_sequences/$pgver/metric.sql mydb
```

NB! Note that there might not be an exact Postgres version match for helper definitions - then replace *$pgver* with the previous available version number below your server's Postgres version number.

NB! Also note that as of v1.8.1 some helpers definition SQL-s scripts (like for "get_stat_statements") will inspect also the "search_path" and by default **will not install into schemas that have PUBLIC CREATE privileges**, like the "public" schema by default has!

Also when rolling out helpers make sure the *search_path* is at defaults or set so that it's also accessible for the monitoring role as currently neither helpers nor metric definition SQL-s don't assume any particualar schema and depend on the *search_path* including everything needed.

For more detailed statistics (OS monitoring, table bloat, WAL size, etc) it is recommended to install also all other helpers found from the */etc/pgwatch2/metrics/00_helpers* folder or do it automatically by using the *rollout_helper.py* script found in the *00_helpers* folder.

As of v1.6.0 though helpers are not needed for Postgres-native metrics (e.g. WAL size) if a privileged user (superuser or *pg_monitor* GRANT) is used, as pgwatch2 now supports having 2 SQL definitions for each metric - "normal / unprivileged" and "privileged" / "superuser". In the file system */etc/pgwatch2/metrics* such "privileged" access definitions will have a "_su" added to the file name.

9.4 Automatic rollout of helpers

pgwatch2 can roll out *helpers* also automatically on the monitored DB. This requires superuser privileges and a configuration attribute for the monitored DB. In YAML config mode it's called *is_superuser*, in Config DB *md_is_superuser*, in the Web UI one can tick the "Auto-create helpers" checkbox and for *ad-hoc* mode there are the *–adhoc-create-helpers* / *PW2_ADHOC_CREATE_HELPERS* flags.

After the automatic rollout it's still generally recommended to remove the superuser privileges from the monitoring role, which now should have GRANT-s to all automatically created helper functions. Note though that all created helpers will not be immediately usable as some are for special purposes and need additional dependencies.

A hint: if it can be foreseen that a lot of databases will be created on some instance (generally not a good idea though) it might be a good idea to roll out the helpers directly in the *template1* database - so that all newly created databases will get them automatically.

9.5 PL/Python helpers

PostgreSQL in general is implemented in such a way that it does not know too much about the operation system that it is running on. This is a good thing for portability but can be somewhat limiting for monitoring, especially when there is no *system monitoring* framework in place or the data is not conveniently accessible together with metrics gathered from Postgres. To overcome this problem, users can also choose to install *helpers* extracting OS metrics like CPU, RAM usage, etc so that this data is stored together with Postgres-native metrics for easier graphing / correlation / alerting. This also enable to be totally independent of any System Monitoring tools like Zabbix, etc, with the downside that everything is gathered over Postgres connections so that when Postgres is down no OS metrics can be gathered also. Since v1.8.4 though the latter problem can be reduced for local "push" based setups via the "–direct-os-stats" option plus according metrics configuration (e.g. the "full" preset).

Note though that PL/Python is usually disabled by DB-as-a-service providers like AWS RDS for security reasons.

```
# first install the Python bindings for Postgres
apt install postgresql-plpython3-XY
# yum install postgresqlXY-plpython3

psql -c "CREATE EXTENSION plpython3u"
psql -f /etc/pgwatch2/metrics/00_helpers/get_load_average/9.1/metric.sql mydb

# psutil helpers are only needed when full set of common OS metrics is wanted
apt install python3-psutil
psql -f /etc/pgwatch2/metrics/00_helpers/get_psutil_cpu/9.1/metric.sql mydb
psql -f /etc/pgwatch2/metrics/00_helpers/get_psutil_mem/9.1/metric.sql mydb
psql -f /etc/pgwatch2/metrics/00_helpers/get_psutil_disk/9.1/metric.sql mydb
psql -f /etc/pgwatch2/metrics/00_helpers/get_psutil_disk_io_total/9.1/metric.sql mydb
```

Note that we're assuming here that we're on a modern Linux system with Python 3 as default. For older systems Python 3 might not be an option though, so you need to change *plpython3u* to *plpythonu* and also do the same replace inside the code of the actual helper functions! Here the *rollout_helper.py* script with it's `--python2` flag can be helpful again.

9.6 Notice on using metric fetching helpers

- Starting from Postgres v10 helpers are mostly not needed (only for PL/Python ones getting OS statistics) - there are available some special monitoring roles like "pg_monitor", that are exactly meant to be used for such cases where we want to give access to all Statistics Collector views without any other "superuser behaviour". See here for documentation on such special system roles. Note that currently most out-of-the-box metrics first rely on the helpers as v10 is relatively new still, and only when fetching fails, direct access with the "Privileged SQL" is tried.
- For gathering OS statistics (CPU, IO, disk) there are helpers and metrics provided, based on the "psutil" Python package… but from user reports seems the package behaviour differentiates slightly based on the Linux distro / Kernel version used, so small adjustments might be needed there (e.g. to remove a non-existent column). Minimum usable Kernel version required is 3.3. Also note that SQL helpers functions are currently defined for Python 3, so for older Python 2 you need to change the `LANGUAGE plpython3u` part.
- When running the gatherer locally, i.e. having a "push" based configuration, the metric fetching helpers are not mostly not needed as superuser can be used in a safe way and starting from v1.8.4 one can also enable the **–direct-os-stats** parameter to signal that we can fetch the data for the default "psutil_*" metrics directly from OS counters. If direct OS fetching fails though, the fallback is still to try via PL/Python wrappers.
- In rare cases when some "helpers" have been installed, and when doing a binary PostgreSQL upgrade at some later point in time via *pg_upgrade*, this could result in error messages thrown. Then just drop those failing helpers on the "to be upgraded" cluster and re-create them after the upgrade process.

9.7 Running with developer credentials

As mentioned above, helper / wrapper functions are not strictly needed, they just provide a bit more information for unprivileged users - thus for developers with no means to install any wrappers as superuser, it's also possible to benefit from pgwatch2 - for such use cases e.g. the "unprivileged" preset metrics profile and the according "DB overview Unprivileged / Developer" dashboard are a good starting point as it only assumes existence of *pg_stat_statements* (which should be available by all cloud providers).

9.8 Different *DB types* explained

When adding a new "to be monitored" entry a *DB type* needs to be selected. Following types are available:

postgres Monitor a single database on a single Postgres instance. When using the Web UI and the "DB name" field is left empty, there's as a one time operation where all non-template DB names are fetched, prefixed with "Unique name" field value and added to monitoring (if not already monitored). Internally monitoring always happens "per DB" not "per cluster" though.

postgres-continuous-discovery Monitor a whole (or subset of DB-s) of Postgres cluster / instance. Host information without a DB name needs to be specified and then the pgwatch2 daemon will periodically scan the cluster and add any found and not yet monitored DBs to monitoring. In this mode it's also possible to specify regular expressions to include/exclude some database names.

pgbouncer Use to track metrics from PgBouncer's "SHOW STATS" command. In place of the Postgres "DB name" the name of the PgBouncer "pool" to be monitored must be inserted.

pgpool Use to track joint metrics from Pgpool2's *SHOW POOL_NODES* and *POOL_PROCESSES* commands. Pgpool2 from version 3.0 is supported.

patroni Patroni is a HA / cluster manager for Postgres that relies on a DCS (Distributed Consensus Store) to store it's state. Typically in such a setup the nodes come and go and also it should not matter who is currently the master. To make it easier to monitor such dynamic constellations pgwatch2 supports reading of cluster node info from all supported DCS-s (etcd, Zookeeper, Consul), but currently only for simpler cases with no security applied (which is actually the common case in a trusted environment).

patroni-continuous-discovery As normal *patroni* DB type but all DB-s (or only those matching the regex if any provided) are monitored.

patroni-namespace-discovery Similar to *patroni-continuous-discovery* but all Patroni scopes (clusters) of an ETCD namespace are automatically monitored. Optionally regexes on database names still apply if provided.

NB! All "continuous" modes expect access to "template1" or "postgres" databasess of the specified cluster to determine the database names residing there.

CHAPTER 10

Monitoring managed cloud databases

Although all cloud service providers offer some kind of built-in instrumentation and graphs, they're mostly rather conservative in this are not to consume extra server resources and not to overflow and confuse beginners with too much information. So for advanced troubleshooting it might make sense to gather some additional metrics on your own, especially given that you can also easily add custom business metrics to pgwatch2 using plain SQL, for example to track the amount of incoming sales orders. Also with pgwatch2 / Grafana you have more freedom on the visual representation side and access to around 30 prebuilt dashboards and a lot of freedom creating custom alerting rules.

The common denominator for all managed cloud services is that they remove / disallow dangerous or potentially dangerous functionalities like file system access and untrusted PL-languages like Python - so you'll lose a small amount of metrics and "helper functions" compared to a standard on-site setup described in the *previous chapter*. This also means that you will get some errors displayed on some preset dashboards like "DB overview" and thus will be better off using a dashboard called "DB overview Unprivileged" tailored specially for such a use case.

pgwatch2 has been tested to work with the following managed database services:

10.1 Google Cloud SQL for PostgreSQL

- No Python / OS helpers possible. OS metrics can be integrated in Grafana though using the Stackdriver data source
- "pg_monitor" system role available
- pgwatch2 default preset name: "gce"
- Documentation: https://cloud.google.com/sql/docs/postgres

NB! To get most out pgwatch2 on GCE you need some additional clicks in the GUI / Cloud Console "Flags" section to enable some common PostgreSQL monitoring parameters like *track_io_timing* and *track_functions*.

10.2 Amazon RDS for PostgreSQL

- No Python / OS helpers possible. OS metrics can be integrated in Grafana though using the CloudWatch data source
- "pg_monitor" system role available
- pgwatch2 default preset names: "rds", "aurora"
- Documentation:

  https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_PostgreSQL.html https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.AuroraPostgreSQL.html

NB! Note that the AWS Aurora PostgreSQL-compatible engine is missing some additional metrics compared to normal RDS.

10.3 Azure Database for PostgreSQL

- No Python / OS helpers possible. OS metrics can be integrated in Grafana though using the Azure Monitor data source
- "pg_monitor" system role available
- pgwatch2 default preset name: "azure"
- Documentation: https://docs.microsoft.com/en-us/azure/postgresql/

NB! Surprisingly on Azure some file access functions are whitelisted, thus one can for example use the "wal_size" metrics.

NB2! By default Azure has **pg_stat_statements** not fully activated by default, so you need to enable it manually or via the API. Documentation link here.

10.4 Aiven for PostgreSQL

The Aiven developer documentation contains information on how to monitor PostgreSQL instances running on the Aiven platform with pgwatch2.

CHAPTER 11

Metric definitions

Metrics are named SQL queries that return a timestamp and pretty much anything else you find useful. Most metrics have many different query text versions for different target PostgreSQL versions, also optionally taking into account primary / replica state and as of v1.8 also versions of installed extensions.

```
-- a sample metric
SELECT
  (extract(epoch from now()) * 1e9)::int8 as epoch_ns,
  extract(epoch from (now() - pg_postmaster_start_time()))::int8 as postmaster_uptime_
↪s,
  case when pg_is_in_recovery() then 1 else 0 end as in_recovery_int;
```

Correct version of the metric definition will be chosen automatically by regularly connecting to the target database and checking the Postgres version, recovery state, and if the monitoring user is a superuser or not. For superusers some metrics have alternate SQL definitions (as of v1.6.2) so that no "helpers" are needed for Postgres-native Stats Collector infos. Using superuser accounts for remote monitoring is of course not really recommended.

There's a good set of pre-defined metrics & metric configs provided by the pgwatch2 project to cover all typical needs, but when monitoring hundreds of hosts you'd typically want to develop some custom *Preset Configs* or at least adjust the metric fetching intervals according to your monitoring goals.

Some things to note about the built-in metrics:

- Only a half of them are included in the *Preset configs* and are ready for direct usage. The rest need some extra extensions or privileges, OS level tool installations etc. To see what's possible just browse the sample metrics.
- Some builtin metrics are marked to be only executed when server is a primary or conversely, a standby. The flags can be inspected / set on the Web UI Metrics tab or in YAML mode by suffixing the metric definition with "standby" or "master". Note that starting from v1.8.1 it's also possible to specify completely alternative monitoring configurations, i.e. metric-interval pairs, for the "standby" (recovery) state - by default the same set of metrics are used for both states.
- There are a couple of special preset metrics that have some non-standard behaviour attached to them:

  change_events The "change_events" built-in metric, tracking DDL & config changes, uses internally some other "*_hashes" metrics which are not meant to be used on their own. Such metrics are described also accordingly on the Web UI /metrics page and they should not be removed.

recommendations When enabled (i.e. interval > 0), this metric will find all other metrics starting with "reco_*" and execute those queries. The purpose of the metric is to spot some performance, security and other "best practices" violations. Users can add new "reco_*" queries freely.

server_log_event_counts This enables Postgres server log "tailing" for errors. Can't be used for "pull" setups though unless the DB logs are somehow mounted / copied over, as real file access is needed. See the *Log parsing* chapter for details.

instance_up For normal metrics there will be no data rows stored if the DB is not reachable, but for this one there will be a 0 stored for the "is_up" column that under normal operations would always be 1. This metric can be used to calculate some "uptime" SLA indicator for example.

11.1 Defining custom metrics

For defining metrics definitions you should adhere to a couple of basic concepts:

- Every metric query should have an "epoch_ns" (nanoseconds since epoch column to record the metrics reading time. If the column is not there, things will still work but server timestamp of the metrics gathering daemon will be used, some a small loss (assuming intra-datacenter monitoring with little lag) of precision occurs.
- Queries can only return text, integer, boolean or floating point (a.k.a. double precision) Postgres data types. Note that columns with NULL values are not stored at all in the data layer as it's a bit bothersome to work with NULLs!
- Column names should be descriptive enough so that they're self-explanatory, but not over long as it costs also storage
- Metric queries should execute fast - at least below the selected *Statement timeout* (default 5s)
- Columns can be optionally "tagged" by prefixing them with "tag_". By doing this, the column data will be indexed by the InfluxDB / Postgres giving following advantages:
  - Sophisticated auto-discovery support for indexed keys/values, when building charts with Grafana.
  - Faster queries for queries on those columns.
  - Less disk space used for repeating values in InfluxDB. Thus when you're for example returning some longish and repetitive status strings (possible with Singlestat or Table panels) that you'll be looking up by some ID column, it might still make sense to prefix the column with "tag_" to reduce disks space.
  - If using InfluxDB storage, there needs to be at least one tag column, identifying all rows uniquely, is more than on row can be returned by the query.
- All fetched metric rows can also be "prettyfied" with any custom static key-value data, per host. To enable use the "Custom tags" Web UI field for the monitored DB entry or "custom_tags" YAML field. Note that this works per host and applies to all metrics.
- For Prometheus the numerical columns are by default mapped to a Value Type of "Counter" (as most Statistics Collector columns are cumulative), but when this is not the case and the column is a "Gauge" then according column attributes should be declared. See below section on column attributes for details.
- NB! For Prometheus all text fields will be turned into tags / labels as only floats can be stored!

Adding and using a custom metric:

For *Config DB* based setups:

1. Go to the Web UI "Metric definitions" page and scroll to the bottom.
2. Fill the template - pick a name for your metric, select minimum supported PostgreSQL version and insert the query text and any extra attributes if any (see below for options). Hit the "New" button to store.

3. Activate the newly added metric by including it in some existing *Preset Config* (listed on top of the page) or add it directly in JSON form, together with an interval, into the "Custom metrics config" filed on the "DBs" page.

For *YAML* based setups:

1. Create a new folder for the metric under "/etc/pgwatch2/metrics". The folder name will be the metric's name, so choose wisely.
2. Create a new subfolder for each "minimally supported Postgres version* and insert the metric's SQL definition into a file named "metric.sql". NB! Note the "minimally supported" part - i.e. if your query will work from version v9.0 to v13 then you only need one folder called "9.0". If there was a breaking change in the internal catalogs at v9.6 so that the query stopped working, you need a new folder named "9.6" that will be used for all versions above v9.5.
3. Activate the newly added metric by including it in some existing *Preset Config* (/etc/pgwatch2/metrics/preset-configs.yaml) or add it directly to the YAML config "custom_metrics" section.

FYI - another neat way to quickly test if the metric can be successfully executed on the "to be monitored" DB is to launch pgwatch2 in *ad-hoc mode*:

```
pgwatch2-daemon \
  --adhoc-config='{"my_new_metric": 10}' --adhoc-conn-str="host=mytestdb␣
↪dbname=postgres" \
  --datastore=postgres --pg-metric-store-conn-str=postgresql://... \
  --metrics-folder2=/etc/pgwatch2/metrics --verbose=info
```

11.2 Metric attributes

Since v1.7 behaviour of plain metrics can be extended with a set of attributes that will modify the gathering in some way. The attributes are stored in YAML files called *metric_attrs.yaml" in a metrics root directory or in the *metric_attribute* Config DB table.

Currently supported attributes:

is_instance_level Enables caching, i.e. sharing of metric data between various databases of a single instance to reduce load on the monitored server.

statement_timeout_seconds Enables to override the default 'per monitored DB' statement timeouts on metric level.

metric_storage_name Enables dynamic "renaming" of metrics at storage level, i.e. declaring almost similar metrics with different names but the data will be stored under one metric. Currently used (for out-of-the box metrics) only for the 'stat_statements_no_query_text' metric, to not to store actual query texts from the "pg_stat_statements" extension for more security sensitive instances.

extension_version_based_overrides Enables to "switch out" the query text from some other metric based on some specific extension version. See 'reco_add_index' for an example definition.

disabled_days Enables to "pause" metric gathering on specified days. See metric_attrs.yaml for "wal" for an example.

disabled_times Enables to "pause" metric gathering on specified time intervals. e.g. "09:00-17:00" for business hours. Note that if time zone is not specified the server time of the gather daemon is used. NB! disabled_days / disabled_times can also be defined both on metric and host (host_attrs) level.

For a sample definition see here.

11.3 Column attributes

Besides the *_tag* column prefix modifier, it's also possible to modify the output of certain columns via a few attributes. It's only relevant for Prometheus output though currently, to set the correct data types in the output description, which is generally considered a nice-to-have thing anyways. For YAML based setups this means adding a "column_attrs.yaml" file in the metric's top folder and for Config DB based setup an according "column_attrs" JSON column should be filled via the Web UI.

Supported column attributes:

prometheus_ignored_columns Columns to be discarded on Prometheus scrapes.

prometheus_gauge_columns Describe the mentioned output columns as of TYPE *gauge*, i.e. the value can change any time in any direction. Default TYPE for pgwatch2 is *counter*.

prometheus_all_gauge_columns Describe all columns of metrics as of TYPE *gauge*.

11.4 Adding metric fetching helpers

As mentioned in *Helper Functions* section, Postgres knows very little about the Operating System that it's running on, so in some (most) cases it might be advantageous to also monitor some basic OS statistics together with the PostgreSQL ones, to get a better head start when troubleshooting performance problems. But as setup of such OS tools and linking the gathered data is not always trivial, pgwatch2 has a system of *helpers* for fetching such data.

One can invent and install such *helpers* on the monitored databases freely to expose any information needed (backup status etc) via Python, or any other PL-language supported by Postgres, and then add according metrics similarly to any normal Postgres-native metrics.

CHAPTER 12

The Admin Web UI

If using pgwatch2 in the centrally managed *Config DB* way, for easy configuration management (adding databases to monitoring, adding metrics) there is a small Python Web application bundled making use of the CherryPy Web-framework.

For mass configuration changes the Web UI has some buttons to disable / enable all hosts for example, but one could technically also log into the configuration database and change the *pgwatch2.monitored_db* table directly.

Besides managing the metrics gathering configurations, the two other useful features for the Web UI would be the possibility to look at the logs of the single components and to verify that metrics gathering is working on the "Stat Statements Overview" page, which will contact the metrics DB (only Postgres and InfluxDB supported) and present some stats summaries.

Default port: **8080**

Sample screenshot of the Web UI:

12.1 Web UI security

By default the Web UI is not secured - anyone can view and modify the monitoring configuration. If some security is needed though it can be enabled:

- HTTPS

  `--ssl, --ssl-cert, --ssl-key, --ssl-certificate-chain` or `PW2_WEBSSL, PW2_WEBCERT, PW2_WEBKEY, PW2_WEBCERTCHAIN`

- Password protection

  `--no-anonymous-access, --admin-user, --admin-password` or `PW2_WEBNOANONYMOUS, PW2_WEBUSER, PW2_WEBPASSWORD`

- Hiding some possibly sensitive information

  `--no-component-logs, --no-stats-summary` or `PW2_WEBNOCOMPONENTLOGS, PW2_WEBNOSTATSSUMMARY`

- Password encryption for the role used for fetching metrics

  `--aes-gcm-keyphrase, --aes-gcm-keyphrase-file` or `PW2_AES_GCM_KEYPHRASE, PW2_AES_GCM_KEYPHRASE_FILE`

  Note that standard *LibPQ .pgpass files* can also be used so there's no requirement to store any passwords in pgwatch2 config DB. Also note that when enabling password encryption, the same key needs to be presented also for the gatherer.

NB! For security sensitive environments make sure to always deploy password protection together with SSL, as it uses a standard cookie based techniques vulnerable to snooping / MITM attacks.

12.2 Exposing the component logs

When using the Docker images, internal component logs (Postgres, Influx, Grafana, Go daemon, Web UI itself) are exposed via the "/logs" endpoint. If this is not wanted set the PW2_WEBNOCOMPONENTLOGS env. variable. Note that if a working "/logs" endpoint is desired also in custom setup mode (non-docker) then some actual code changes are needed to specify where logs of all components are situated - see top of the pgwatch2.py file for that.

CHAPTER 13

Dashboarding and alerting

13.1 Grafana intro

To display the gathered and stored metrics the pgwatch2 project has decided to rely heavily on the popular Grafana dashboarding solution. This means only though that it's installed in the default Docker images and there's a set of predefined dashboards available to cover most of the metrics gathered via the *Preset Configs*.

This does not mean though that Grafana is in any way tightly coupled with project's other components - quite the opposite actually, one can use any other means / tools to use the metrics data gathered by the pgwatch2 daemon.

Currently there are around 30 preset dashboards available for PostgreSQL data sources and about 25 for InfluxDB. The reason why InfluxDB (originally the only supported metrics storage DB) has less, is the fact that the custom query language is just much more limiting compared to standard SQL so that it makes it hard to build more complex dashboards. Due to that nowadays, if metric gathering volumes are not a problem, we recommend using Postgres storage for most users.

Note though that most users will probably want to always adjust the built-in dashboards slightly (colors, roundings, etc) , so that they should be taken only as examples to quickly get started. NB! Also note that in case of changes it's not recommended to change the built-in ones, but use the *Save as* features - this will allow later to easily update all the dashboards *en masse* per script, without losing any custom user changes.

Links:

Built-in dashboards for PostgreSQL (TimescaleDB) storage

Built-in dashboards for InfluxDB storage

Screenshots of pgwatch2 default dashboards

The online Demo site

13.2 Alerting

Alerting is very conveniently also supported by Grafana in a simple point-and-click style - see here for the official documentation. In general all more popular notification services are supported and it's pretty much the easiest way to

quickly start with PostgreSQL alerting on a smaller scale. For enterprise usage with hundreds of instances it's might get too "clicky" though and there are also some limitations - currently you can set alerts only on Graph panels and there must be no variables used in the query so you cannot use most of the pre-created pgwatch2 graphs, but need to create your own.

Nevertheless, alerting via Grafana is s a good option for lighter use cases and there's also a preset dashboard template named "Alert Template" from the pgwatch2 project to give you some ideas on what to alert on.

Note though that alerting is always a bit of a complex topic - it requires good understanding of PostgreSQL operational metrics and also business criticality background infos, so we don't want to be too opinionated here and it's up to the users to implement.

CHAPTER 14

Sizing recommendations

- Min 1GB of RAM is required for a Docker setup using Postgres to store metrics, for InfluxDB it should be at least 2GB.

  The gatherer alone needs typically less than 50 MB if the metrics store is online. Memory consumption will increase a lot when the metrics store is offline though, as then metrics are cached in RAM in ringbuffer style up to a limit of 10k data points (for all databases) and then memory consumption is dependent on how "wide" are the metrics gathered.

- Storage requirements vary a lot and are hard to predict.

  2GB of disk space should be enough though for monitoring a single DB with "exhaustive" *preset* for 1 month with InfluxDB storage. 1 month is also the default metrics retention policy for Influx running in Docker (configurable). Depending on the amount of schema objects - tables, indexes, stored procedures and especially on number of unique SQL-s, it could be also much more. With Postgres as metric store multiply it with ~5x, but if disk size reduction is wanted for PostgreSQL storage then best would be to use the TimescaleDB extension - it has built-in compression and disk footprint is on the same level with InfluxDB, while retaining full SQL support.

  To determine disk usage needs for your exact setup, there's also a *Test Data Generation* mode built into the collector. Using this mode requires two below params, plus also the usual *Ad-hoc mode* connection string params.

  Relevant params: *–testdata-days, –testdata-multiplier or PW2_TESTDATA_DAYS, PW2_TESTDATA_MULTIPLIER*

- A low-spec (1 vCPU, 2 GB RAM) cloud machine can easily monitor 100 DBs in "exhaustive" settings (i.e. almost all metrics are monitored in 1-2min intervals) without breaking a sweat (<20% load).

- A single InfluxDB node should handle thousands of requests per second but if this is not enough, pgwatch2 also supports writing into a secondary "mirror" InfluxDB, to distribute at least read load evenly. If more than two are needed one should look at InfluxDB Enterprise, Graphite, PostgreSQL with streaming replicas for read scaling or for example Citus for write scaling.

- When high metrics write latency is problematic (e.g. using a DBaaS across the atlantic) then increasing the default maximum batching delay of 250ms usually gives good results.

  Relevant params: *–batching-delay-ms / PW2_BATCHING_MAX_DELAY_MS*

- Note that when monitoring a very large number of databases, it's possible to "shard" / distribute them between many metric collection instances running on different hosts, via the *group* attribute. This requires that some hosts have been assigned a non-default *group* identifier, which is just a text field exactly for this sharding purpose.

  Relevant params: *–group / PW2_GROUP*

CHAPTER 15

Technical details

Here are some technical details that might be interesting for those who are planning to use pgwatch2 for critical monitoring tasks or customize it in some way.

- Dynamic management of monitored databases, metrics and their intervals - no need to restart/redeploy

  Config DB or YAML / SQL files are scanned every 2 minutes (by default, changeable via –servers-refresh-loop-seconds) and changes are applied dynamically. As common connectivity errors also also handled, there should be no need to restart the gatherer "for fun". Please always report issues which require restarting.

- There are some safety features built-in so that monitoring would not obstruct actual operation of databases
  - Up to 2 concurrent queries per monitored database (thus more per cluster) are allowed
  - Configurable statement timeouts per DB
  - SSL connections support for safe over-the-internet monitoring (use "-e PW2_WEBSSL=1 -e PW2_GRAFANASSL=1" when launching Docker)
  - Optional authentication for the Web UI and Grafana (by default freely accessible)

- Instance-level metrics caching

  To further reduce load on multi-DB instances, pgwatch2 can cache the output of metrics that are marked to gather only instance-level data. One such metric is for example "wal", and the *metric attribute* is "is_instance_level". Caching will by activated only for *continuous DB types*, and to a default limit of up to 30 seconds (changeable via the –instance-level-cache-max-seconds param).

CHAPTER 16

Security aspects

16.1 General security information

Security can be tightened for most pgwatch2 components quite granularly, but the default values for the Docker image don't focus on security though but rather on being quickly usable for ad-hoc performance troubleshooting, which is where the roots of pgwatch2 lie.

Some points on security:

- Starting from v1.3.0 there's a non-root Docker version available (suitable for OpenShift)
- The administrative Web UI doesn't have by default any security. Configurable via env. variables.
- Viewing Grafana dashboards by default doesn't require login. Editing needs a password. Configurable via env. variables.
- InfluxDB has no authentication in Docker setup, so one should just not expose the ports when having concerns.
- Dashboards based on the "stat_statements" metric (Stat Statement Overview / Top) expose actual queries.

  They should be "mostly" stripped of details though and replaced by placeholders by Postgres, but if no risks can be taken such dashboards (or at least according panels) should be deleted. Or as an alternative the "stat_statements_no_query_text" and "pg_stat_statements_calls" metrics could be used, which don't store query texts in the first place.
- Safe certificate connections to Postgres are supported as of v1.5.0

  According *sslmode* (verify-ca, verify-full) and cert file paths need to be specified then on Web UI "/dbs" page or in the YAML config.
- Encryption / decryption of connection string passwords stored in the config DB or in YAML config files

  By default passwords are stored in plaintext but as of v1.5 it's possible to use an encryption passphrase, or a file with the passphrase in it, via *–aes-gcm-keyphrase* / *–aes-gcm-keyphrase-file* or *PW2_AES_GCM_KEYPHRASE* / *PW2_AES_GCM_KEYPHRASE_FILE* parameters. If using the Web UI to store connection info, the same encryption key needs to be specified for both the Web UI and the gatherer. If using YAML configs then encrypted passwords can be generated using the *–aes-gcm-password-to-encrypt* flag for embedding in YAML.

Note that although pgwatch2 can handle password security, in many cases it's better to still use the standard LibPQ *.pgpass* file to store passwords.

16.2 Launching a more secure Docker container

Some common sense security is built into default Docker images for all components but not actived by default. A sample command to launch pgwatch2 with following security "checkpoints" enabled:

1. HTTPS for both Grafana and the Web UI with self-signed certificates
2. No anonymous viewing of graphs in Grafana
3. Custom user / password for the Grafana "admin" account
4. No anonymous access / editing over the admin Web UI
5. No viewing of internal logs of components running inside Docker
6. Password encryption for connect strings stored in the Config DB

```
docker run --name pw2 -d --restart=unless-stopped \
  -p 3000:3000 -p 8080:8080 \
  -e PW2_GRAFANASSL=1 -e PW2_WEBSSL=1 \
  -e PW2_GRAFANANOANONYMOUS=1 -e PW2_GRAFANAUSER=myuser -e PW2_GRAFANAPASSWORD=mypass␣
↪\
  -e PW2_WEBNOANONYMOUS=1 -e PW2_WEBNOCOMPONENTLOGS=1 \
  -e PW2_WEBUSER=myuser -e PW2_WEBPASSWORD=mypass \
  -e PW2_AES_GCM_KEYPHRASE=qwerty \
  cybertec/pgwatch2-postgres
```

NB! For custom installs it's up to the user though. A hint - Docker *launcher* files can also be inspected to see which config parameters are being touched.

CHAPTER 17

Long term installations

For long term pgwatch2 setups the main challenge is to keep the software up-to-date to guarantee stable operation and also to make sure that all DB-s are under monitoring.

17.1 Keeping inventory in sync

Adding new DBs to monitoring and removing those shut down, can become a problem if teams are big, databases are many, and it's done per hand (common for on-premise, non-orchestrated deployments). To combat that, the most typical approach would be to write some script or Cronjob that parses the company's internal inventory database, files or endpoints and translate changes to according CRUD operations on the *pgwatch2.monitored_db* table directly.

One could also use the Web UI page (pseudo) *API* for that purpose, if the optional Web UI component has been deployed. See here for an usage example - but direct database access is of course more flexible.

If pgwatch2 configuration is kept in YAML files, it should be also relatively easy to automate the maintenance as the configuration can be organized so that one file represent a single monitoring entry, i.e. the *–config* parameter can also refer to a folder of YAML files.

17.2 Updating the pgwatch2 collector

The pgwatch2 metrics gathering daemon is the core component of the solution alas the most critical one. So it's definitely recommended to update it at least once per year or minimally when some freshly released Postgres major version instances are added to monitoring. New Postgres versions don't necessary mean that something will break, but you'll be missing some newly added metrics, plus the occasional optimizations. See the ref:*upgrading chapter <upgrading>* for details, but basically the process is very similar to initial installation as the collector doesn't have any state on its own - it's just on binary program.

17.3 Metrics maintenance

Metric definition SQL-s are regularly corrected as suggestions / improvements come in and also new ones are added to cover latest Postgres versions, so would make sense to refresh them 1-2x per year.

If using a YAML based config, just installing newer pre-built RPM / DEB packages will do the trick automatically (built-in metrics at */etc/pgwatch2/metrics* will be refreshed) but for Config DB based setups you'd need to follow a simple process described *here*.

17.4 Dashboard maintenance

Same as with metrics, also the built-in Grafana dashboards are being actively updates, so would make sense to refresh them occasionally also. The bulk delete / import scripts can be found here or you could also manually just re-import some dashboards of interest from JSON files in */etc/pgwatch2/grafana-dashboards* folder or from Github.

NB! The *delete_all_old_pw2_dashes.sh* script deletes all pgwatch2 built-in dashboards so you should take some extra care when you've changed them. In general it's a good idea not to modify the preset dashboards too much, but rate use the "Save As…" button and rename the dashboards to something else.

FYI - notable new dashboards are usually listed also in release notes and most dashboards also have a sample screenshots available.

17.5 Storage monitoring

In addition to all that you should at least initially periodically monitor the metrics DB size… as it can grow quite a lot (especially when using Postgres for storage) when the monitored databases have hundreds of tables / indexes and if a lot of unique SQL-s are used and *pg_stat_statements* monitoring is enabled. If the storage grows too fast, one can increase the metric intervals (especially for "table_stats", "index_stats" and "stat_statements") or decrease the data retention periods via *–pg-retention-days* or *–iretentiondays* params.

CHAPTER 18

Upgrading

The pgwatch2 daemon code doesn't need too much maintenance itself (if you're not interested in new features), but the preset metrics, dashboards and the other components that pgwatch2 relies, like Grafana, are under very active development and get updates quite regularly so already purely from the security standpoint it would make sense to stay up to date.

We also regularly include new component versions in the Docker images after verifying that they work. If using Docker, you could also choose to build your own images any time some new component versions are released, just increment the version numbers in the Dockerfile.

18.1 Updating to a newer Docker version

18.1.1 Without volumes

If pgwatch2 container was started in the simplest way possible without volumes, and if previously gathered metrics are not of great importance, and there are no user modified metric or dashboard changes that should be preserved, then the easiest way to get the latest components would be just to launch new container and import the old monitoring config:

```
# let's backup up the monitored hosts
psql -p5432 -U pgwatch2 -d pgwatch2 -c "\copy monitored_db to 'monitored_db.copy'"

# stop the old container and start a new one ...
docker stop ... && docker run ....

# import the monitored hosts
psql -p5432 -U pgwatch2 -d pgwatch2 -c "\copy monitored_db from 'monitored_db.copy'"
```

If metrics data and other settings like custom dashboards need to be preserved then some more steps are needed, but basically it's about pulling InfluxDB / Postgres backups and restoring them into the new container - see the take_backup.sh script for an example with InfluxDB storage.

A tip: to make the restore process easier it would already make sense to mount the host folder with the backups in it on the new container with "-v ~/pgwatch2_backups:/pgwatch2_backups:rw,z" when starting the Docker image.

Otherwise one needs to set up SSH or use something like S3 for example. Also note that ports 5432 and 8088 need to be exposed to take backups outside of Docker for Postgres and InfluxDB respectively.

18.1.2 With volumes

To make updates a bit easier, the preferred way to launch pgwatch2 should be to use Docker volumes for each individual component - see the *Installing using Docker* chapter for details. Then one can just stop the old container and start a new one, re-using the volumes.

With some releases though, updating to newer version might additionally still require manual rollout of Config DB *schema migrations scripts*, so always check the release notes for hints on that or just go to the "pgwatch2/sql/migrations" folder and execute all SQL scripts that have a higher version than the old pgwatch2 container. Error messages like will "missing columns" or "wrong datatype" will also hint at that, after launching with a new image. FYI - such SQL "patches" are generally not provided for metric updates, nor dashboard changes and they need to be updated separately.

18.2 Updating without Docker

For a custom installation there's quite some freedom in doing updates - as components (Grafana, InfluxDB, PostgreSQL) are loosely coupled, they can be updated any time without worrying too much about the other components. Only "tightly coupled" components are the pgwatch2 metrics collector, config DB and the optional Web UI - if the pgwatch2 config is kept in the database. If YAML based approach (see details *here*) is used, then things are even more simple - the pgwatch2 daemon can be updated any time as YAML schema has default values for everything and there are no other "tightly coupled" components like the Web UI.

18.3 Updating Grafana

The update process for Grafana looks pretty much like the installation so take a look at the according *chapter*. If using Grafana's package repository it should happen automatically along with other system packages. Grafana has a built-in database schema migrator, so updating the binaries and restarting is enough.

18.4 Updating Grafana dashboards

There are no update or migration scripts for the built-in Grafana dashboards as it would break possible user applied changes. If you know that there are no user changes, then one can just delete or rename the existing ones in a bulk matter and import the latest JSON definitions. See *here* for some more advice on how to manage dashboards.

18.5 Updating the config / metrics DB version

Database updates can be quite complex, with many steps, so it makes sense to follow the manufacturer's instructions here.

For InfluxDB typically something like that is enough though (assuming Debian based here):

```
influxd version # check current version
INFLUX_LATEST=$(curl -so- https://api.github.com/repos/influxdata/influxdb/releases/
↪latest \
                    | jq .tag_name | grep -oE '[0-9\.]+')
wget https://dl.influxdata.com/influxdb/releases/influxdb_${INFLUX_LATEST}_amd64.deb
sudo dpkg -i influxdb_${INFLUX_LATEST}_amd64.deb
```

For PostgreSQL one should distinguish between minor version updates and major version upgrades. Minor updates are quite straightforward and problem-free, consisting of running something like:

```
apt update && apt install postgresql
sudo systemctl restart postgresql
```

For PostgreSQL major version upgrades one should read through the according release notes (e.g. here) and be prepared for the unavoidable downtime.

18.6 Updating the pgwatch2 schema

This is the pgwatch2 specific part, with some coupling between the following components - Config DB SQL schema, metrics collector, and the optional Web UI.

Here one should check from the CHANGELOG if pgwatch2 schema needs updating. If yes, then manual applying of schema diffs is required before running the new gatherer or Web UI. If no, i.e. no schema changes, all components can be updated independently in random order.

Assuming that we initially installed pgwatch2 version v1.6.0, and now the latest version is 1.6.2, based on the release notes and SQL diffs we need to apply the following files:

```
psql -f /etc/pgwatch2/sql/config_store/migrations/v1.6.1-1_patroni_cont_
↪discovery.sql pgwatch2
psql -f /etc/pgwatch2/sql/config_store/migrations/v1.6.2_superuser_metrics.
↪sql pgwatch2
```

18.7 Updating the metrics collector

Compile or install the gatherer from RPM / DEB / tarball packages. See the *Custom installation* chapter for details.

If using a SystemD service file to auto-start the collector then you might want to also check for possible updates on the template there - */etc/pgwatch2/startup-scripts/pgwatch2.service*.

18.8 Updating the Web UI

Update the optional Python Web UI if using it to administer monitored DB-s and metric configs. The Web UI was not included in the pre-built packages of older pgwatch2 versions as deploying self-contained Python that runs on all platforms is not overly easy. If Web UI is started directly on the Github sources (*git clone && cd webpy && ./web.py*) then it is actually updated automatically as CherryPy web server monitors the file changes. If there were some breaking schema changes though, it might stop working and needs a restart after applying schema "diffs" (see above).

If using a SystemD service file to auto-start the Web UI then you might want to also check for possible updates on the template there - */etc/pgwatch/webpy/startup-scripts/pgwatch2-webui.service*.

18.9 Updating metric definitions

In the YAML mode you always get new SQL definitions for the built-in metrics automatically when refreshing the sources via Github or pre-built packages, but with Config DB approach one needs to do it manually. Given that there are no user added metrics, it's simple enough though - just delete all old ones and re-insert everything from the latest metric definition SQL file.

```
pg_dump -t pgwatch2.metric pgwatch2 > old_metric.sql  # a just-in-case backup
psql  -c "truncate pgwatch2.metric" pgwatch2
psql -f /etc/pgwatch2/sql/config_store/metric_definitions.sql pgwatch2
```

NB! If you have added some own custom metrics be sure not to delete or truncate them!

CHAPTER 19

Kubernetes

A basic Helm chart is available for installing pgwatch2 to a Kubernetes cluster. The corresponding setup can be found in *./openshift_k8s/helm-chart*, whereas installation is done via the following commands:

```
cd openshift_k8s
helm install -f chart-values.yml pgwatch2 ./helm-chart
```

Please have a look at openshift_k8s/helm-chart/values.yaml to get additional information of configurable options.

CHAPTER 20

Indices and tables

- genindex
- modindex
- search