SQLTrack

Joachim Folz

May 07, 2024

CONTENTS

SQLTrack is a library and set of tools to track and analyze your machine learning experiments.

We made SQLTrack, because using alternatives like MLflow Tracking or Sacred never felt like we truly owned our data, and we were limited by what these tools "allowed" us to do.

CHAPTER

ONE

"JUST LET ME PROGRAM IT!"

Clicking through GUIs can be time consuming and tedious, especially when you have to do so repeatedly, yet results are limited to whatever the GUI can do. Code is the most powerful user interface, why not use it to analyze your experiments?

This is what SQLTrack allows you to do. Easily and reliably collect and store experiment data in a well known, and most importantly highly accessible format: *a SQL database*, minus all the annoying boilerplate.

SQLTrack provides a simple yet powerful schema for experiments, runs, and metrics that you can extend to suit your needs, along with some tools to set up the database, and store and retrieve data with few lines.

CHAPTER

TWO

A MINIMAL EXAMPLE

Here is a minimal example for how to track an experiment:

```
import sqltrack
from sqltrack.commands import setup

def main():
    client = sqltrack.Client()
    setup(client)
    experiment = sqltrack.Experiment(client, name="Minimal")
    run = experiment.get_run()
    with run.track():
        for epoch in range(90):
            metrics = {"loss": 7**(1/(epoch+1))}
            run.add_metrics(metrics, step=epoch)

if __name__ == "__main__":
    main()
```

And this is a minimal example for how to retrieve data:

```
import sqltrack
from sqltrack.pandas import query_dataframe

def main():
    client = sqltrack.Client()
    metrics = query_dataframe(client, "SELECT * FROM metrics")
    print(metrics)

if __name__ == "__main__":
    main()
```

Which outputs:

```
    run_id  step  progress      loss
0        1     0       0.0  7.000000
1        1     1       0.0  2.645751
2        1     2       0.0  1.912931
3        1     3       0.0  1.626577
4        1     4       0.0  1.475773
..     ...   ...       ...       ...
85       1    85       0.0  1.022885
```

(continues on next page)

(continued from previous page)

```
86       1    86       0.0  1.022619
87       1    87       0.0  1.022359
88       1    88       0.0  1.022105
89       1    89       0.0  1.021857

[90 rows x 4 columns]
```

You can now do whatever you wish with the `metrics` stored in this `pandas.DataFrame`. See *Analyzing experiments* for more advanced examples.

CHAPTER

THREE

CONTENTS

3.1 Getting started

3.1.1 Installation

The default install with minimal dependencies (optimized for containerized environments) provides core functionality located in the toplevel *sqltrack* package. It enables tracking experiments and working with the database:

```
pip install sqltrack
```

To use some of the convenience functions for analysis later, install with the full option:

```
pip install sqltrack[full]
```

This enables use of, e.g., the *sqltrack.notebook* and *sqltrack.pandas* modules.

SQLite for local use

If you just want to try SQLTrack or simply use it *locally*, you can use SQLite instead of the default PostgreSQL engine. Simply tell SQLTrack to use the `sqlite` engine with one of these methods:

- Create your *Client* with `client = Client(engine="sqlite")`.
- Put `engine=sqlite` in your `sqltrack.conf`.
- Set `SQLTRACK_DSN_ENGINE=sqlite` in your environment.

Note: We *strongly* advise against using SQLite for anything other than local testing and analysis. It is not safe to use unless all processes are located on the same machine. You should migrate to PostgreSQL as soon as possible. See the SQLite Frequently Asked Questions and How To Corrupt An SQLite Database File for detailed explanations.

Installing PostgreSQL

Linux

Plase follow the install instructions for Linux and your distribution, though beware of potentially outdated package names and versions.

If you cannot use the package manager, e.g., because you lack sudo privileges, or want to use a different version, you can try pgenv.

MacOS

Plase follow the install instructions for MacOS.

Windows

Plase follow the install instructions for Windows.

User and database creation

If you're not comfortable working with PostgreSQL (or databases in general) yet, we recommend to use the `sqltrack create` tool to generate the necessary commands for you:

```
$ sqltrack create CREATE ROLE alice NOSUPERUSER NOCREATEDB
NOCREATEROLE INHERIT LOGIN; CREATE DATABASE alice OWNER
alice;
```

If PostgreSQL is running locally, you can simply pipe the output into psql:

```
sqltrack create | sudo -u postgres psql
```

See the *`sqltrack create`* page for details on how to use it.

Hint: Other than passwords, PostgreSQL provides a multitude of authentication methods. The ident method could be particularly interesting for users on computer clusters. Once a compatible service (e.g. oidentd) is installed a compute node, users can connect to PostgreSQL without credentials, as if it was running locally. This is safe as long as logins are controlled by a central authentication service, such as LDAP.

3.1.2 Setup the database

Databases need to be setup so SQLTrack can store your experiment data. The *`sqltrack setup`* tool does this for you:

```
$ sqltrack setup (NEW) base.sql
```

User, host, dbname, and schema as parameters given on the command line take priority, but you can also define environment variables `SQLTRACK_DSN_<PARAM>` to set them. More info on available parameters can be found here. Finally, most convenient is probably to store them in a config file. The default path is `./sqltrack.conf`.

```
user=<USER>
host=<HOST>
dbname=<DATABASE>
schema=<SCHEMA>
```

See *Configuration* for a more in depth explanation of how to configure SQLTrack.

Those SQL script files you created earlier? This is where you use them. Run the setup command with them, e.g. `sqltrack setup v001.sql`. This creates the base schema and updates it with your definitions.

3.1.3 Track an experiment

```python
import sqltrack
from sqltrack.commands import setup

def main():
    client = sqltrack.Client()
    setup(client)
    experiment = sqltrack.Experiment(client, name="Minimal")
    run = experiment.get_run()
    with run.track():
        for epoch in range(90):
            metrics = {"loss": 7**(1/(epoch+1))}
            run.add_metrics(metrics, step=epoch)

if __name__ == "__main__":
    main()
```

3.1.4 Analyzing results

This is where it's up to you. We recommend Jupyter Lab to interact with the database, but plain Jupyter or alternatives like Plotly Dash work well too. Look at the examples directory in our repository to get some ideas. But really, you're the experimenter, you know best what to do with your data.

To find out how SQLTrack can help you create tools that are exactly right for you, head on over to our guide on how to *analyze experiments*.

3.2 Core concepts

Currently SQLTrack supports PostgreSQL through the psycopg3 driver, as well as `SQLite` for local use.

We decided against using an ORM like SQLAlchemy, as it would add layers of indirection between our users and their data. Ideally we would use standard SQL and let users bring their own Python DB-API 2.0 compatible driver, but that would mean we lose access to advanced features like indexable JSONB columns.

3.2.1 Experiments

Experiments are the top level concept in SQLTrack. They are stored in the `experiments` table. It has the following columns:

`id`	unique ID	`int`
`name`	unique name	`str`
`comment`	free text comment	`str`
`tags`	tags, stored as a dict `{"tag": True}`	`dict` (JSONB)
`extras`	extra bits of data that is not tags	`dict` (JSONB)

Tags are stored as a dict, where keys are the text tags and all values are `True`. This definition makes it easy to query, add, and remove tags. You can store anything that isn't a tag as an extra, though it must be JSON serializable.

Experiment links

SQLTrack supports adding links between experiments, e.g., to express a predecessor/successor relationships. Experiment links are stored in the `experiment_links` table. It has the following columns:

`from_id`	ID of the origin experiment	`int`
`kind`	what kind of link this is	`str`
`to_id`	ID of the target experiment	`int`

3.2.2 Runs

Runs are individual, tracked executions of some code, like a job on an HPC cluster. Each run belongs to one experiment and stores metadata about the exeuction. Runs are stored in the `runs` table. It has the following columns:

`id`	unique ID	`int`
`experiment_id`	ID of the experiment this run belongs to	`int`
`status`	text status of the run, defaults to `"PLANNED"`	`str`
`time_created`	Creation timestamp, must not be null	`datetime`
`time_started`	Start timestamp	`datetime`
`time_updated`	Last update timestamp	`datetime`
`comment`	free text comment	`str`
`tags`	tags, stored as a dict `{"tag": True}`	`dict` (JSONB)
`args`	arguments for this run	`dict` (JSONB)
`env`	environment variables for this run	`dict` (JSONB)
`extras`	extra bits of data that don't fit anywhere else	`dict` (JSONB)

Run links

Like links between experiments, SQLTrack supports adding links between runs. Common uses could be to express that one run resumes or repeats another run which failed, or a pre-training/fine-tuning relationship. Run links are stored in the `run_links` table. It has the following columns:

`from_id`	ID of the experiment this run belongs to	`int`
`kind`	what kind of link this is	`str`
`to_id`	ID of the target run	`int`

3.2.3 Metrics

Metrics are measurements taken during run execution. They are stored in the `metrics` table. It has the following columns:

`run_id`	ID of the run these metrics belong to	`int`
`step`	Integer step of the experiment	`int`
`progress`	Run progress expressed as a real number	`float`

Importantly, metrics measured at the same point in time are stored together in the same row. You must set `step` or `progress` (or both) to define this point. By convention, `step` refers to an absolute value like iterations, and `progress` refers to a completion percentage.

Defining metrics

After initializing the database with the base schema, the `metrics` table doesn't contain any columns to store metrics yet. We recommend users add the required columns before starting experiments, however, *run_add_metrics()* (and *Run.add_metrics()*) will try to infer types and attempt to add columns if necessary. To manually define metrics, create a script to add columns. A script that defines columns for timing, loss, and accuracy in train, validation, and test phases could look like this:

```
BEGIN;

ALTER TABLE metrics ADD COLUMN train_start TIMESTAMP WITH TIME ZONE;
ALTER TABLE metrics        ADD COLUMN train_end TIMESTAMP WITH TIME ZONE;
ALTER TABLE metrics        ADD COLUMN train_loss FLOAT;
ALTER TABLE metrics        ADD COLUMN train_top1 FLOAT;
ALTER TABLE metrics        ADD COLUMN train_top5 FLOAT;
ALTER TABLE metrics        ADD COLUMN val_start TIMESTAMP WITH TIME ZONE;
ALTER TABLE metrics        ADD COLUMN val_end TIMESTAMP WITH TIME ZONE;
ALTER TABLE metrics        ADD COLUMN val_loss FLOAT;
ALTER TABLE metrics        ADD COLUMN val_top1 FLOAT;
ALTER TABLE metrics        ADD COLUMN val_top5 FLOAT;
ALTER TABLE metrics        ADD COLUMN test_start TIMESTAMP WITH TIME ZONE;
ALTER TABLE metrics        ADD COLUMN test_end TIMESTAMP WITH TIME ZONE;
ALTER TABLE metrics        ADD COLUMN test_loss FLOAT;
ALTER TABLE metrics        ADD COLUMN test_top1 FLOAT;
ALTER TABLE metrics        ADD COLUMN test_top5 FLOAT;

END;
```

Call this script `v001.sql` or similar, and use it with *`sqltrack setup`*:

```
sqltrack setup v001.sql
```

If you want to make changes later, simply create a `v002.sql` and run setup again.

Automatic types with PostgreSQL

With the PostgreSQL engine, SQLTrack follows the adaptation strategy of psycopg. This provides seamless translation for commonly used built-in types like `int`, `float`, `str` etc.

Automatic types with SQLite

With SQLite, the following types are inferred:

- `int`: `INTEGER`
- `float`: `REAL`
- `str`: `TEXT`
- `bytes`: `BLOB`
- `datetime.date`: `DATE`
- `datetime.datetime`: `TIMESTAMP WITH TIME ZONE`
- `datetime.timedelta`: `INTERVAL`
- `Jsonb`: `JSONB`

Is one column per metric not inefficient?

You might ask why you should use columns for your metrics, because that seems annoying and wasteful compared to a normalized name+value approach like what MLflow uses (one row per value with run ID, metric name, and timestamp). But don't worry, because PostgreSQL is smart and doesn't actually store `NULL` values. It only stores values that are not `NULL` and uses a bitmap per row to keep track of them.

By contrast, each row has a fixed size header of ~23 bytes and MLflow uses one row per metric value. Since we store many metric values in a row we can afford really large bitmaps to track those `NULL` values before we come out worse. Even if you only ever store one metric per row, the break even point is over 200 columns.

3.2.4 Full schema

If you wish to dive deeper into the inner workings of SQLTrack, you can find the full base schema for PostgreSQL at sqltrack/engines/postgres/base.sql, and for SQLite at sqltrack/engines/sqlite/base.sql. They contain the definitions of the corresponding tables `experiments`, `experiment_links`, `runs`, `run_links`, and `metrics`, as well as indices and some additional definitions.

3.3 Configuration

SQLTrack configuration currently only refers to the *Client*. The `engine` parameter (either `postgres` or `sqlite`) determines which *engine* to use. Engine-specific parameters are passed along, see *SQLite* and *PostgreSQL* below for details. Configuration can be given either directly as dictionary and `kwargs`, or as path to a config file.

If a dictionary is given, `kwargs` can add or overwrite existing parameters. This option is intended to hardcode the configuration, thus no other sources of parameters are considered.

Otherwise, SQLTrack will attempt to load config parameters from a file. If no path is given, the environment variable `SQLTRACK_CONFIG_PATH` is used. If neither path nor `SQLTRACK_CONFIG_PATH` are given, some *Default config locations* are checked. Besides the config file, the default config, environment variables, and `kwargs` are also considered. Conflicts between parameters set through different methods are resolved in the following order, from lowest to highest priority:

1. *Default config*
2. Config file (if any)
3. `SQLTRACK_DSN_<PARAM>` environment variables; see *get_env_config()* for details
4. `kwargs` given to *Client*

The config loading mechanism is implemented in *load_config()*.

3.3.1 Default config

SQLTrack uses the following default configuration:

```
DEFAULT_CONFIG = {
    "engine": "postgres",
    "dbpath": "sqltrack.db",
    "user": getpass.getuser(),
    "dbname": getpass.getuser(),
}
```

`getpass.getuser()` is essentially a more robust version of `$USER` that also considers some fancy alternatives ways to determine the actual username.

3.3.2 Config file syntax

SQLTrack uses `configparser` to parse config files. The `[DEFAULT]` section header is implicitly added for convenience. Here's an example config file:

```
user = alice
dbname = bob
host = postgres
```

3.3.3 Default config locations

Client accepts configuration either directly as dictionary or via `kwargs`, or a path to a config file. The path can also be set through the `SQLTRACK_CONFIG_PATH` environment variable.

If no path is given explicitly, these default location are checked, in this order:

1. `./sqltrack.conf`
2. `$XDG_CONFIG_HOME/sqltrack/sqltrack.conf`
3. `%APPDATA%/sqltrack/sqltrack.conf`
4. `$HOME/.config/sqltrack/sqltrack.conf`
5. `$HOME/sqltrack.conf`

The first path that exists is used. If none of them exists SQLTrack reverts to the default configuration. You can use *find_config_file()* to verify the correct file is used.

3.3.4 SQLite

Set `engine=sqlite` to use SQLite. The *SQLite engine* accepts one parameter `dbpath`, the path where the database is stored. It defaults to `"sqltrack.db"`.

3.3.5 PostgreSQL

Set `engine=postgres` (the default) to use PostgreSQL. Parameters accepted by the *PostgreSQL engine* are listed in the libpq documentation. Some of the most important parameters are:

- host
- dbname
- user
- password
- passfile

`libpq` also looks for certain environment variables

The PostgreSQL engine also accept a convenience parameter `schema`. It is an alias for `options=--search_path=<SCHEMA>`, meaning identifiers (names of tables, etc.) resolve to the given schema.

3.4 Analyzing experiments

To follow the provided examples, make sure you installed SQLTrack with the `full` optional deptendencies: `pip install sqltrack[full]`.

Important: Whichever platform you choose and what you do with it is up to you. These examples are meant to guide and inspire.

3.4.1 Jupyter notebooks

Our first example is a Jupyter notebook (see examples/notebook.ipynb) that recreates an mlflow-like experience through SQLTrack, but with some important improvements, like displaying the best metric value for a run instead of the latest.

Jupyter notebooks

This notebook demonstrates how to use SQLTrack to replicate and improve on some features found in other tools, like the mlflow UI. You can find the original notebook at examples/notebook.ipynb. The data displayed here is generated by examples/generate_experiment_data.py.

We will display lists of experiments and runs, show metrics and plot them, and finally compare settings between different runs.

Let's start by setting up our environment. We use itables to display interactive tables and Plotly for plots. The config for our SQLTrack *Client* is loaded from `./sqltrack.conf`.

[1]:
```
import itables
import plotly.offline as po
import sqltrack
import sqltrack.notebook as stn

itables.init_notebook_mode()
# tell plotly not to embed javascript it into the notebook
# javascript is loaded form the notebook server instead
# this drastically reduces the filesize of notebooks
po.init_notebook_mode(connected=True)
# add sqltrack CSS
stn.init_notebook_mode()

client = sqltrack.Client()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

Experiments

First, we will show all experiments in the database. We use *query_dataframe* to run a query against the database and pack all returned rows into a Pandas DataFrame. *format_dataframe* applies some HTML & CSS formatting to the columns. E.g., timestamps are converted to human-friendly relative time strings with the humanize library. You can hover over the text to see the actual time, and thanks to some invisible text it also sorts correctly.

[2]:
```
from sqltrack.pandas import query_dataframe
from sqltrack.notebook import format_dataframe

with client.cursor() as cursor:
    experiments = query_dataframe(cursor, "SELECT * FROM experiments")
itables.show(format_dataframe(experiments))
```

```
<IPython.core.display.HTML object>
```

Runs

Now let's try something similar with a more complex query. We will display all runs, but only show the columns we want to see. This is also where one major improvement over the mlflow UI can be made. We join the `runs` and `metrics` table to display metrics for the step that achieve maximum top-1 accuracy. For performance reasons, mlflow always display the latest metric value, which is not necessarily the best. We also see the PostgreSQL syntax to interactiv with JSONB columns, e.g. `tags ? 'marked'` to check if a key is present and `env->'GIT_COMMIT'` to extract a value. Check their docs for more details on all the different things you can do with JSONB columns.

We again use *format_dataframe* for some nice formatting, but additionally tell it that we want `val_top1` formatted as a percentage.

[3]:
```
import itables

from sqltrack.pandas import query_dataframe
from sqltrack.notebook import format_dataframe
from sqltrack.notebook import format_float
from sqltrack.notebook import format_percentage

with client.cursor() as cursor:
    runs = query_dataframe(cursor, """
        SELECT DISTINCT ON (id)
            tags ? 'marked' as " ",
            id,
            status as s,
            time_updated as updated,
            time_updated - time_started as runtime,
            step,
            progress,
            val_top1,
            val_loss,
            args->'lr' as lr,
            env->'SOURCE' as source,
            env->'GIT_COMMIT' as commit,
            env->'SLURM_JOB_PARTITION' as partition,
            tags
        FROM runs LEFT JOIN metrics ON id = run_id
        ORDER BY id, val_top1 DESC
    """).sort_values("id", ascending=False).reset_index(drop=True)

mapping = {"val_top1": format_percentage}
itables.show(format_dataframe(runs, mapping))
```

```
<IPython.core.display.HTML object>
```

Metrics

Finally, let's look at metrics. By now you should be familiar with querying the database and displaying the result as table. So let's add another concept that SQLTrack supports: links.

Both experiments and runs can have named links to other experiments and runs. In our example we claim that run 523473 "resumes" run 523459. If you flip to page 5 in the table, you can see that metrics for run 523459 end at step (epoch) 44. From step 45 onwards metrics are from run 523473. We use the "resumes" link to merge both ids to 523459 to make it clear that this should have been one run, and make it easier to plot.

[4]:
```
from sqltrack.pandas import query_dataframe

run_ids = (523459,)

with client.cursor() as cursor:
    metrics = query_dataframe(cursor, """
        SELECT
            COALESCE((
                SELECT to_id FROM run_links WHERE run_id = from_id AND kind = 'resumes'),
                run_id
            ) as merged_id,
            *
        FROM metrics
        WHERE run_id = ANY(%(run_ids)s) OR run_id IN (
            SELECT from_id
            FROM run_links
            WHERE to_id = ANY(%(run_ids)s)
        );
    """, {"run_ids": list(run_ids)}).sort_values("step")
itables.show(format_dataframe(metrics))
```

```
<IPython.core.display.HTML object>
```

Plots

This is a fairly standard afair for notebooks. We use Plotly to create a plot from the DataFrame we created in the previous step. Nothing fancy here, just showing that mlflow-style plots are easy to create. However, here we have full control over what is plotted and how it looks.

[5]:
```
import plotly.graph_objects as go

fig = go.Figure(layout=dict(title="Loss curves", xaxis=dict(title="epoch"),␣
↪yaxis=dict(title="loss")))
for run_id, run_metrics in metrics.groupby("merged_id"):
    fig.add_trace(go.Scatter(x=run_metrics["step"], y=run_metrics["train_loss"], name=f"
↪{run_id} train loss"))
    fig.add_trace(go.Scatter(x=run_metrics["step"], y=run_metrics["val_loss"], name=f"
↪{run_id} val loss"))
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Since this is a Jupyter notebook, we can of course add Markdown cells wherever we like to provide additional commentary. We could, for example, discuss something interesting we saw in the plot we just made. This way, instead of clicking through a tracking UI and taking notes elsewhere, you naturally create a sort of interactive "report" of your progress.

Run comparison

Finally, we'll compare some runs. Again, the query might look a bit scary, but it's really mostly the selection of columns.

The tricky part is to unpack our `args` and `env` JSONB columns into individual columns in the DataFrame. We find that this works best outside of SQL with the `json_normalize` method of the DataFrame.

Most noteworthy is that we join with the metrics table twice. Once to get the best metrics, and again to get prograss and average epoch time. These are just some examples of the kind of flexibility SQLTrack provides to its users.

[6]:
```
import pandas as pd
from sqltrack.notebook import format_timedelta, format_datetime_relative

def compare_runs(*run_ids):
    with client.cursor() as cursor:
        runs = query_dataframe(cursor, """
            SELECT
                runs.id as id,
                runs.status as status,
                runs.time_started as started,
                runs.time_updated as updated,
                runs.time_updated - time_started as runtime,
                runs.env->'SLURM_JOBID' as jobid,
                exp.id as experiments_id,
                exp.name as experiments_name,
                runs.comment as comment,
                runs.tags as tags,
                runs.tags ? 'marked' as marked,
                metrics.step,
                metrics.progress,
                metrics.epoch_time,
                best_metrics.step as best_step,
                best_metrics.val_top1,
                best_metrics.val_loss,
                runs.args as args,
                runs.env as env
            FROM runs
                JOIN experiments AS exp ON experiment_id = exp.id
                LEFT JOIN (
                    SELECT DISTINCT ON (run_id) *
                    FROM metrics
                    ORDER BY run_id, val_top1 DESC
                ) AS best_metrics ON runs.id = best_metrics.run_id
                LEFT JOIN (
                    SELECT
                        run_id,
                        MAX(step) AS step,
                        MAX(progress) AS progress,
```

(continues on next page)

(continued from previous page)

```
                        AVG(train_end - train_start) AS epoch_time
                    FROM metrics
                    GROUP BY run_id
                ) AS metrics ON runs.id = metrics.run_id
            WHERE runs.id = ANY(%(run_ids)s)
            ORDER BY runs.id ASC
        """, {"run_ids": list(run_ids)}).sort_values("id").reset_index(drop=True)

    args = pd.json_normalize(runs['args'])
    env = pd.json_normalize(runs['env'])
    runs = runs.drop(['args', 'env'], axis=1)
    runs = runs.join([args, env])
    runs = runs.set_index('id')

    columnDefs=[{"className": "dt-center", "targets": list(range(1, len(runs.index)+1))},
                { "targets": "_all", "createdCell": itables.JavascriptFunction(
                    """
                    function (td, cellData, rowData, row, col) {
                        if (col>0 && !rowData.slice(1).every( (val, i, arr) => val === arr[0] )){
                            $(td).css('color', 'OrangeRed')
                        }
                    }
                    """)
    }]

    mapping = { "started": format_datetime_relative,
                "best_val_top1": format_percentage,
                "epoch_time": format_timedelta,
            }

    itables.show(
        format_dataframe(runs, mapping).T,
        classes="cell-max-width-15em",
        columnDefs=columnDefs,
        paging=False,
        dom="frt",
    )

compare_runs(523497, 523473, 523459, 1)
```

```
<IPython.core.display.HTML object>
```

[Optional] Self-signed SSL certificate

You can create a SSL self-signed certificate to use Juypter Lab with HTTPS:

```
openssl req -x509 -newkey rsa:4096 -keyout jupyter.key -out jupyter.crt -sha256 -days␣
↪365 -nodes
```

Start Jupyter Lab with your certificate:

```
jupyter-lab [options...] --certfile jupyter.crt --keyfile jupyter.key
```

3.4.2 Plotly Dash

Todo: Create a custom UI with Dash.

3.5 Database management

Todo: (Optional) Don't let public connect to your database:

```
REVOKE CONNECT ON DATABASE ${USERNAME} FROM PUBLIC;
```

Give a different user access to your database/schema:

```
GRANT CONNECT ON DATABASE ${DATABASE} TO ${USERNAME};
GRANT USAGE ON SCHEMA ${SCHEMA} TO ${USERNAME};
GRANT SELECT ON ALL TABLES IN SCHEMA ${SCHEMA} TO ${USERNAME};
```

3.6 Argument parsing

SQLTrack includes argument parsing functions based on docopt-ng, a fork of the original docopt that is actively maintained.

Instead of writing an argument parser in code, docopt parses help texts in POSIX syntax to know what arguments exist, whether they are switches or parameters, etc. For our purposes the help texts are extracted from docstrings in the main script file. This means we don't need to run the script to obtain its arguments, so we can add them to the database even if the run has not started yet, e.g., because it is in the queue of a batch scheduling system.

Here's a simple example:

```
from sqltrack.args import docopt_main

@docopt_main
def main(args):
    """
    usage: example [options] [--learning-rate N...]

```

(continues on next page)

(continued from previous page)

```
    options:
        -h --help               Print help text.
        --model M               Which model to train [default: resnet18]
        -e N, --epochs N        Number of training epochs [default: 90]
        -b N, --batch-size N    Mini-batch size
        -l R, --learning-rate R  Learning rate  [default: 0.1]
        --amp                   Use AMP (Automatic Mixed Precision)
    """
    print(args)
    print(args.epochs)

# This will run main a second time with AMP forced on and set epochs to 360.
# You should not do this in practice, since docopt_main already calls the
# main function, but this is only a silly example anyways.
main({"amp": True, "epochs": 360})
```

Our main function is decorated with *docopt_main*, which parses the command line arguments defined in the docstring and immediately calls `main` (with the usual `if __name__ == "__main__"` guard). For the sake of completeness we also call `main({"amp": True, "epochs": 360})` ourselves, which is a bit silly, since you would normally run the main function only once, but illustrates how to call the decorated function from code.

```
$ python examples/argument_parsing.py -e 180
{'amp': False,
 'batch_size': None,
 'epochs': 180,
 'help': False,
 'learning_rate': ['0.1'],
 'model': 'resnet18'}
180
{'amp': True,
 'batch_size': None,
 'epochs': 360,
 'help': False,
 'learning_rate': ['0.1'],
 'model': 'resnet18'}
360
```

The output tells us that the `args` object passed to `main` is a dictionary. More precisely it is of type ParsedOptions, a dictionary subclass that can be accessed via attributes, like you would with an `argparse.Namespace` object. In the example above, we print the batch size with `print(args.batch_size)`.

Warning: One caveat of *docopt_main* is that it immediately calls the decorated function, so it must be defined after everything else in your script. If this is not something you want, you can use *docopt_arguments* instead and add the `if __name__ == "__main__"` guard yourself as usual. It does the same thing, but doesn't call the decorated function.

3.6.1 Argument types

POSIX help texts do not define types for arguments, so docopt simply returns parsed values as strings. While this is 100% safe, it is quite annoying to use in practice and challenges the main reason why we opted to use docopt in the first place: to parse arguments without running code.

We opted to include a – what we believe to be – reasonable mechanism to guess types in SQLTrack. First, we try a suffix match of the argument name with a number of explicit conversion functions. If all these fail we finally try to convert values to number types (integer, float, complex) and finally JSON.

Here's an overview of all conversions that are attempted by default:

- `name` matches `*int` → `int`
- `name` matches `*float` → `float`
- `name` matches `*complex` → `complex`
- `name` matches `*path` → `pathlib.Path`
- `name` matches `*json` → `json.loads()`
- `name` matches `*str` → `str`
- try `int`
- try `float`
- try `complex`
- try `json.loads()`

The final trial and error stage is fixed. While you can append new conversions (or replace existing ones, without changing the order) with *`sqltrack.args.register_conversion()`*, we recommend you don't, as your changes to the conversion logic cannot be replicated without running your code.

Hint: You can use suffix matching to avoid edge cases. E.g., to avoid the conversion of `--version 3.0` to float, use `--versionstr 3.0` instead.

3.6.2 Limitations

Multiple values

Many argument parsers (like `argparse`) allow arguments with multiple values. One argument with three values could be represented on the command line as `--arg 1 2 3`. Docopt does not support this, as it always expects argument-value pairs if the argument is not a simple switch.

You can instead specify that an argument may be repeated in the usage part of the help text like so:

```
Usage: example [options] [--arg VALUE...]
```

The equivalent command line would then be `--arg 1 --arg 2 --arg 3`. Values for repeatable arguments are passed as lists, even if there is only one value.

Another alternative that avoids repeating the argument name is to use the conversion from JSON built into SQLTrack, e.g., `--arg [1, 2, 3]`. In this case you should not specify the argument as repeatable, or else the result would be a nested list `[[1, 2, 3]]`.

3.7 Why we made SQLTrack

Alternative title: "a rant about experiment tracking".

For some reason tracking experiments is still hard today. It shouldn't be. Here's our thoughts on the topic. Just bullet points for now, since we can't be bothered to ask ChatGPT to write it for us.

- How to keep track of your experiment results?
  - Experiment data is precious
  - Structure is necessary to compare experiment runs
  - Ad-hoc methods (spreadsheets et al.) can work well, but become cumbersome for hundreds/thousands of runs
  - Need automated solutions
- Are hosted tracking services a good idea?
  - Many solutions (like wandb) to choose from
  - Typically a free-tier for individuals, but very costly for teams
  - Service can go down, or become slow (*will* happen right before a paper deadline)
  - Service provider can change conditions, effectively hold data hostage (e.g., introducing a monthly tracking time limit for free users)
  - Self-hosing is a must!
- What self-hosted solutions are there?
  - Basically mlflow
    - Basic use cases are easy enough
    - Terse, but OK documentation
    - Comes with a first-party web GUI
  - Alternatives
    - Sacred
      - OG tracking & reproducibility tool
      - Cool features like code versioning and automatic arameter parsing
      - GUI frontends are available, but none of them do what we need
    - ploomber-engine
      - Inspiration for our solution, but lots of limitations
      - Flat hierarchy with just experiments, no runs
      - Only one set of metrics per experiment
      - Relies on magic to detect metrics from global scope
    - MLTRAQ
      - Similar to our solution
      - DB schema with one row per experiment and deeply nested JSON columns
      - Every time a metric is added the whole row needs to be rewritten

        - Should be a performance nightmare
    - Are we missing something? Let us know
- Our issues with mlflow
    - No concept of authentication, users, permissions, … need to do everything yourself
    - By default all tracked parameters & environment are display in GUI
        - Need to select relevant columns
        - URL is used to store settings, including selected columns
        - Selection stops working if you have too many columns, because URL is too long
    - Cannot change the order of columns in tables
    - Run overview always shows lastest metric value
        - Schema makes aggregation over metric tables slow
        - A separate table with the lastest value per run is used as a workaround
        - Other aggretations could be done similarly, but it is difficult to add them and this doesn't scale
    - Experiments/runs cannot be linked to other experiments/runs
        - Pre-training? Fine-tuning?
    - Graphs are too small
    - So much clicking, let us program our analyses already!
- Our solution: just use SQL!
    - SQL is almost 50 years old and still relevant, so it has to have done something right
        - Mature ecosystem with great tools and tons of great resources to learn
        - A lot of people know it already
        - Fine-grained user privilege controls down to single tables
    - Experiment data is not actually that complex, easy to map to relational DBs, especially with modern features like JSON columns
    - You know your experiments, just define your metrics as columns, avoid mlflow performance problems
    - Build whichever analyses you like, display them wherever, e.g. as reports with notes in Jupyter Notebooks
    - Trivial conversion from SQL to Pandas Dataframe
    - SQL + Pandas + Jupyter = insane flexibility **FOR FREE**

3.8 API reference

3.8.1 sqltrack package

class sqltrack.**Client**(*config: dict | str | Path | None = None*, ***kwargs*)

Bases: `object`

Creates and manages database connections (currently `psycopg.Connection` or `sqlite3.Connection`).

For simple queries, use the *`execute()`* method:

```
client = Client(...)
client.execute(...)
```

Use a client as context manager to obtain a database connection object:

```
client = Client(...)
with client as conn
    with conn.cursor() as cursor:
        ...
```

If all you need is a cursor, you can obtain one directly from a client with the *cursor()* method:

```
client = Client(...)
with client.cursor() as cursor:
    ...
```

Connection parameters are given as `kwargs`. Use the `engine` parameter to `"postgres"` (default) or `sqlite`. For SQLite, only `dbpath` is relevant. It defaults to `sqltrack.db`. For Postgres, common options are:

- `user`: defaults to USER env var
- `dbname`: defaults to `user`
- `host`
- `schema`: a shorthand for setting the `search_path` option.

For the full list of available parameters, see: https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-PARAMKEYWORDS

Parameters passed from Python take priority, but they may also be passed as environment variables `SQLTRACK_DSN_<PARAM>` (e.g., `SQLTRACK_DSN_USER`), or loaded from a config file, by default `./sqltrack.conf`.

SQLTrack classes and functions will connect to the database as required. Nested contexts (`with` blocks) reuse the same connection (reentrant), so they can be used to avoid connecting to the database multiple times over a short period. The connection is closed and any uncomitted changes are comitted when the outermost `with` block ends. E.g., the following snippet will connect only once:

```
def do_queries(client, ...):
    with client.cursor() as cursor:
        cursor.execute(...)
        ...

client = Client(...)
with client:
    do_queries(client, ...)
    do_queries(client, ...)
    do_queries(client, ...)
```

One caveat of this approach is that all changes in a stack of contexts implicitly happen within the same transaction. All pending changes will be rolled back if an exception is raised. You can avoid this by periodically calling *commit()*.

Parameters

- **config** – Config dict or path to config file, defaults to `SQLTRACK_CONFIG_PATH` environment variable, and finally `./sqltrack.conf`

- **kwargs** – Connection parameters

commit()

Convenience function to call commit on the DB connection. Raises RuntimeError when not connected.

cursor() → DBAPICursor

Connect to the DB and return a cursor. Use in with statement:

```
with client.cursor() as cursor:
    ... cursor things ...
```

The connection is closed and any changes comitted when the with block ends.

execute(*sql*, *parameters=()*)

Convenience function that connects to the DB, if necessary, and executes the given query with optional parameters.

executemany(*sql*, *seq_of_parameters*)

Convenience function that connects to the DB, if necessary, and calls executemany with the given sequence of parameters.

executescript(*sql_script*)

Convenience function that connects to the DB, if necessary, and executes the given script.

rollback()

Convenience function to call rollback on the DB connection. Raises RuntimeError when not connected.

class sqltrack.**Experiment**(*client: Client*, *experiment_id: int | None = None*, *name: str | None = None*)

Bases: object

Helper class to create experiments, as well as runs for experiments.

Note: All methods (except *get_run*) return self to allow chaining calls, e.g., `exp = Experiment(client, id).set_comment("nice").add_tags("tag")`

Parameters

- **client** – Client object to use.
- **experiment_id** – ID of the experiment. May be None if name is given.
- **name** – Name of the experiment. may be None if experiment_id is given.

add_link(*kind: str*, *to_id: int*) → *Experiment*

Add a link to another experiment.

add_tags(**tags: str*) → *Experiment*

Add tags to the experiment.

get_run(*run_id: int | str | None = None*) → *Run*

Get a *Run* object. See its documentation for more details.

remove_link(*kind: str*, *to_id: int*) → *Experiment*

Remove a link to another experiment.

remove_tags(**tags: str*) → *Experiment*

Remove tags from the experiment.

set_comment(*comment: str*) → *Experiment*

Set the experiment comment.

set_name(*name: str*) → *Experiment*

Set the experiment name.

set_tags(**tags: str*) → *Experiment*

Set tags of the experiment.

class sqltrack.**Run**(*client:* Client, *experiment_id: int | None = None*, *run_id: int | str | None = None*)

Bases: object

Helper class to manage runs.

Note: If `run_id` is None, a new run with an unused ID is created.

Note: All methods return self to allow chaining calls, e.g., `run = Run(client, exp_id).set_comment("nice").add_tags("tag")`

Parameters

- **client** – Client object to use.
- **experiment_id** – ID of the experiment. May be None if an existing run_id is given.
- **run_id** – ID of the run. If None an unused ID is chosen. If string then load ID from that environment variable.

add_args(*args: 'auto' | dict | None = 'auto'*) → *Run*

Add parameters to the run. See *run_add_args()* for details.

add_env(*env: 'auto' | dict | None = 'auto'*) → *Run*

Add environment variables to the run. See *run_add_env()* for details.

add_extras(*extras: dict | None = None*) → *Run*

Add extras to the run. See *run_add_extras()* for details.

add_link(*kind: str*, *to_id: int*) → *Run*

Add a link to another run.

add_metrics(*metrics: dict*, *step: int = 0*, *progress: float = 0.0*, *updated: 'auto' | datetime | None = 'auto'*) → *Run*

Add metrics to the run. See *run_add_metrics()* for details.

add_tags(**tags: str*) → *Run*

Add tags to the run. See *run_add_tags()* for details.

remove_args(*args: 'auto' | dict | None = 'auto'*) → *Run*

Remove parameters from the run. See *run_remove_args()* for details.

remove_env(*env: 'auto' | dict | None = 'auto'*) → *Run*

Remove environment variables from the run. See *run_remove_env()* for details.

remove_extras(**tags: str*) → *Run*

Remove extras from the run. See *run_remove_extras()* for details.

remove_link(*kind: str*, *to_id: int*) → *Run*

Remove a link to another run.

remove_tags(**tags: str*) → *Run*

Remove tags from the run. See *run_remove_tags()* for details.

set_args(*args: 'auto' | dict | None = 'auto'*) → *Run*

Set the run parameters. See *run_set_args()* for details.

set_comment(*comment: str*) → *Run*

Set the run comment.

set_created(*dt: 'auto' | datetime | None = 'auto'*) → *Run*

Set time_created for the run. See *run_set_created()* for details.

set_env(*env: 'auto' | dict | None = 'auto'*) → *Run*

Set the run environment. See *run_set_env()* for details.

set_extras(**tags: str*) → *Run*

Set the run extras. See *run_set_extras()* for details.

set_started(*dt: 'auto' | datetime | None = 'auto'*) → *Run*

Set time_started for the run. See *run_set_started()* for details.

set_status(*status: str*) → *Run*

Set the run status. See *run_set_status()* for details.

set_tags(**tags: str*) → *Run*

Set the run tags. See *run_set_tags()* for details.

set_updated(*dt: 'auto' | datetime | None = 'auto'*) → *Run*

Set time_updated for the run. See *run_set_updated()* for details.

start(*terminated='CANCELLED'*, *started: 'auto' | datetime | None = 'auto'*, *updated: 'auto' | datetime | None = 'auto'*) → *Run*

Start the run, setting its status to RUNNING, among other values like `time_started`, depending on parameters.

Parameters

- **terminated** – status in case SIGTERM is received during the run; see also *sqltrack.sigterm*
- **started** – what to do about time_started; See *run_set_started()* for details
- **updated** – what to do about time_updated; See *run_set_updated()* for details

stop(*status='COMPLETED'*, *updated: 'auto' | datetime | None = 'auto'*) → *Run*

Stop the run, setting its status to COMPLETED and `time_started` to now (default), depending on parameters.

Parameters

- **status** – status to set (default: COMPLETED); See *run_set_status()* for details
- **updated** – what to do about time_updated; See *run_set_updated()* for details

track(*normal='COMPLETED'*, *exception='FAILED'*, *interrupt='CANCELLED'*, *terminated='CANCELLED'*, *started: 'auto' | datetime | None = 'auto'*, *updated: 'auto' | datetime | None = 'auto'*) → *Run*

A context manager to track the execution of the run. This is equivalent to calling start and stop separately with the appropriate status value.

Parameters

- **normal** – status in case the run completes normally
- **exception** – status in case an exception occurs
- **interrupt** – status in case SIGINT is received during the run
- **terminated** – status in case SIGTERM is received during the run; see also *sqltrack.sigterm*
- **started** – what to do about time_started; See *run_set_started()* for details
- **updated** – what to do about time_updated; See *run_set_started()* for details

sqltrack.**experiment_add_link**(*client:* Client, *from_id:* *int*, *kind:* *str*, *to_id:* *int*)

Add a link between two experiments.

sqltrack.**experiment_add_tags**(*client:* Client, *experiment_id:* *int*, **tags:* *str*)

Add tags to an experiment.

sqltrack.**experiment_remove_link**(*client:* Client, *from_id:* *int*, *kind:* *str*, *to_id:* *int*)

Remove a link between two experiments.

sqltrack.**experiment_remove_tags**(*client:* Client, *experiment_id:* *int*, **tags:* *str*)

Remove tags from an experiment.

sqltrack.**experiment_set_comment**(*client:* Client, *experiment_id:* *int*, *comment:* *str* | *None*)

Set an experiment comment.

sqltrack.**experiment_set_name**(*client:* Client, *experiment_id:* *int*, *name:* *str*)

Set an experiment name.

sqltrack.**experiment_set_tags**(*client:* Client, *experiment_id:* *int*, **tags:* *str*)

Set tags of an experiment.

sqltrack.**run_add_args**(*client:* Client, *run_id:* *int*, *args:* *'auto'* | *dict* | *None* = *'auto'*)

Add some arguments to an existing run. See *sqltrack.args.detect_args()* for details on detection in auto mode.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **args** – the arguments, may be `'auto'` (detect arguments), `dict`, or None (do nothing)

sqltrack.**run_add_env**(*client:* Client, *run_id:* *int*, *env:* *'auto'* | *dict* | *None* = *'auto'*)

Add some environment variables to an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **env** – the environment, may be `'auto'` (set to `os.environ`), `dict`, or None

sqltrack.**run_add_extras**(*client:* Client, *run_id:* *int*, *extras:* *dict* | *None* = *None*)

Add some extras to an existing run.

Parameters

- **client** – the Client to use

- **run_id** – which existing run to update
- **extras** – the extras, may be `dict`, or None (do nothing)

`sqltrack.`**run_add_link**(*client:* Client, *from_id:* *int*, *kind:* *str*, *to_id:* *int*)

Add a link between two runs.

`sqltrack.`**run_add_metrics**(*client:* Client, *run_id:* *int*, *metrics:* *dict*, *step:* *int* = *0*, *progress:* *float* = *0.0*)

Add metrics to a run.

Important: Either `step` or `progress` need to be a non-zero value to avoid overwriting existing metric values.

Note: If there is no corresponding column for metrics, this function will attempt to create them. See *Defining metrics* for details.

`sqltrack.`**run_add_tags**(*client:* Client, *run_id:* *int*, **tags:* *str*)

Add tags to an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **tags** – tags to add

`sqltrack.`**run_from_env**(*client:* Client) → *Run*

Get the Run object defined by environment variables. The experiment is defined by at least one or both of `SQLTRACK_EXPERIMENT_NAME` and `SQLTRACK_EXPERIMENT_ID`, and optionally `SQLTRACK_RUN_ID`.

`sqltrack.`**run_remove_args**(*client:* Client, *run_id:* *int*, **args:* *str*)

Remove some arguments from an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **args** – names to remove from arguments

`sqltrack.`**run_remove_env**(*client:* Client, *run_id:* *int*, **env:* *str*)

Remove some environment variables from an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **env** – names to remove from env

`sqltrack.`**run_remove_extras**(*client:* Client, *run_id:* *int*, **extras:* *str*)

Remove some extras from an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update

- **extras** – names to remove from extras

sqltrack.**run_remove_link**(*client:* Client, *from_id:* *int*, *kind:* *str*, *to_id:* *int*)

Remove a link between two runs.

sqltrack.**run_remove_tags**(*client:* Client, *run_id:* *int*, **tags:* *str*)

Remove tags from an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **tags** – tags to remove

sqltrack.**run_set_args**(*client:* Client, *run_id:* *int*, *args:* *'auto'* | *dict* | *None* = *'auto'*)

Set run arguments. See *sqltrack.args.detect_args()* for details on detection in auto mode. Nothing is done if no arguments are detected in auto mode.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **args** – the arguments, may be `'auto'` (detect arguments), `dict`, or None (set NULL)

sqltrack.**run_set_comment**(*client:* Client, *run_id:* *int*, *comment:* *str* | *None*)

Set the comment for an existing run.

sqltrack.**run_set_created**(*client:* Client, *run_id:* *int*, *dt:* *'auto'* | *datetime* | *None* = *'auto'*)

Set time_created for an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **dt** – the timestamp, may be `'auto'` (set to now), timezone aware `datetime`, or None (do nothing)

sqltrack.**run_set_env**(*client:* Client, *run_id:* *int*, *env:* *'auto'* | *dict* | *None* = *'auto'*)

Set run environment.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **env** – the environment, may be `'auto'` (set to `os.environ`), `dict`, or None (set to `NULL`)

sqltrack.**run_set_extras**(*client:* Client, *run_id:* *int*, *extras:* *dict* | *None* = *None*)

Set run extras.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **extras** – the environment, `dict`, or None (set to NULL)

sqltrack.**run_set_started**(*client:* Client, *run_id:* *int*, *dt:* *'auto'* | *datetime* | *None* = *'auto'*)

Set time_started for an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **dt** – the timestamp, may be `'auto'` (set to now), timezone aware `datetime`, or None (do nothing)

sqltrack.**run_set_status**(*client:* Client, *run_id:* *int*, *status:* *str*)

Set a run's status.

sqltrack.**run_set_tags**(*client:* Client, *run_id:* *int*, **tags:* *str*)

Set the tags of an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **tags** – the tags

sqltrack.**run_set_updated**(*client:* Client, *run_id:* *int*, *dt:* *'auto'* | *datetime* | *None* = *'auto'*)

Set time_updated for an existing run.

Parameters

- **client** – the Client to use
- **run_id** – which existing run to update
- **dt** – the timestamp, may be `'auto'` (set to now), timezone aware `datetime`, or None (do nothing)

3.8.2 sqltrack.args module

sqltrack.args.**detect_args**(*path:* *Path* | *str* | *None* = *None*, ***kwargs*) → ParsedOptions | None

Try to detect command line arguments using docopt. Docstrings are extracted from the file at the given path. `sys.argv[0]` is used if no path is given.

First, docstrings of functions decorated with *docopt_arguments()* or *docopt_main()* are parsed in the order that they appear, and finally the module docstring. The first set of arguments that is successfully parsed is returned.

Parameters

- **path** – Use docstrings from this file, or `sys.argv[0]` if None
- **kwargs** – extra arguments passed to docopt.docopt()

sqltrack.args.**docopt_arguments**(*f=None*, *main=False*, *main_guard=True*, ***kwargs*)

Decorator to parse command line arguments using docopt. The decorated function must accept one positional argument, e.g.:

```
@docopt_arguments
def main(args):
    ...
```

The function docstring is parsed first, then the module docstring. The first set of successfully parsed arguments is passed to the function. ValueError is raised if parsing fails for all docstrings.

You can provide additional arguments, or override command line arguments by passing a dictionary to the wrapped function: `main({"good": True})`. An empty dictionary (`main({})`) has no effect.

Parameters

- **f** – the decorated function, filled in by the Python
- **main** – if True, immediately run `f` with parsed args
- **main_guard** – if True, guard execution of `f` with `if __name__ == "__main__"`
- **kwargs** – extra arguments passed to docopt.docopt()

sqltrack.args.**docopt_main**(*f=None*, *main_guard=True*, ***kwargs*)

Decorator to parse command line arguments using *docopt <https://github.com/jazzband/docopt-ng>*. This is an alias for:

```
@docopt_arguments(main=True)
def main(args):
    ...
```

Functions decorated like this are immediately executed, so they need to be located at the end of the file after any other definitons. If this is not what you want, use *docopt_arguments()* instead:

```
@docopt_arguments
def main(args):
    ...

if __name__ == "__main__":
    main({})
```

sqltrack.args.**docopt_parse_docstrings**(*docstrings: Iterable[str]*, *simplify_names=True*, ***kwargs*) → ParsedOptions

Use the `docopt()` package to parse arguments based on the POSIX definition of calling syntax in the given docstrings. Arguments of the first successful parsing are returned. Raises ValueError if parsing all given docstrings fails.

Arguments are converted from strings using `convert_arguments()`.

Parameters

- **docstrings** – docstrings to parse, None values are ignored
- **simplify_names** – if True, simplify argument names; See `convert_arguments()` for details
- **kwargs** – extra arguments passed to docopt.docopt()

sqltrack.args.**make_conversion**(*pattern: str*, *func: Callable*)

Create a function that applies the given conversion function to arguments whose names match the given pattern.

Parameters

- **pattern** – A `fnmatch.fnmatch()` pattern
- **func** – A function that accepts one string argument and returns the converted result

sqltrack.args.**register_conversion**(*name: str*, *func: Callable*)

Register a function for automatic argument conversion. Functions must have signature `conversion(name: str, value: str) -> Tuple[object, bool]`. If the conversion

3.8.3 sqltrack.commands package

The SQLTrack command line utility. It makes setting up databases for use with SQLTrack a bit easier.

sqltrack.commands.**main**()

Submodules

sqltrack.commands.create module

The create tool guides users through the creation of PostgreSQL users and databases.

This naturally requires privileged access to PostgreSQL. Rather than requiring these privileges, this tool will never execute any commands. It merely prints suggested commands that users can inspect, modify, and execute as they see fit.

Alternatively, you can pipe its output into a psql to execute the commands directly, e.g., for a locally running PostgreSQL:

sqltrack create | sudo -u postgres psql

If you don't have access to psql, you can also use sqltrack execute:

sqltrack create | sqltrack execute

Regarding passwords:

If `--password` is specified, this tool will try to determine a password for the new user, first from the pgpass file, followed by prompting for it. To avoid printing it in plain text, it will then attempt to encrypt the password. To do so, it must connect to the database. Should this fail, you have two options to remedy the situation:

- add `--plain-password` to allow printing passwords as plain text
- provide credentials via `--config-path` or one of the other ways described here: https://sqltrack.readthedocs.io/en/latest/configuration.html

sqltrack.commands.create.**create**(*config: dict*, *no_user: bool = False*, *no_db: bool = False*, ***kwargs*)

sqltrack.commands.setup module

sqltrack.commands.setup.**setup**(*client:* Client, *scripts: Iterable[str | Path | Tuple[str, str]] = ()*)

Execute SQL scripts to setup (or update) the database. The included `base.sql` script is always executed first. User-defined scripts are run in the given order.

Scripts can be loaded from files, or defined directly as tuples `(name, script)`, where `script` is the SQL code to execute.

A script is never run twice. Whether a script has already been run before is determined by its base filename without any directories. The rest of the path is ignored. Thus `base.sql` cannot be used as filename for user-defined scripts.

Example script with timestamps, loss and accuracies for training, validation, and test phases:

```
BEGIN;

ALTER TABLE metrics
    ADD COLUMN train_start TIMESTAMP WITH TIME ZONE,
    ADD COLUMN train_end TIMESTAMP WITH TIME ZONE,
    ADD COLUMN train_loss FLOAT,
    ADD COLUMN train_top1 FLOAT,
    ADD COLUMN train_top5 FLOAT,
    ADD COLUMN val_start TIMESTAMP WITH TIME ZONE,
    ADD COLUMN val_end TIMESTAMP WITH TIME ZONE,
    ADD COLUMN val_loss FLOAT,
    ADD COLUMN val_top1 FLOAT,
    ADD COLUMN val_top5 FLOAT,
    ADD COLUMN test_start TIMESTAMP WITH TIME ZONE,
    ADD COLUMN test_end TIMESTAMP WITH TIME ZONE,
    ADD COLUMN test_loss FLOAT,
    ADD COLUMN test_top1 FLOAT,
    ADD COLUMN test_top5 FLOAT;

END;
```

Parameters

- **client** – Client to connect to the database
- **scripts** – Paths to SQL scripts or tuples `(name, script)`; executed in the given order

3.8.4 sqltrack.config module

This module contains functions to locate and parse SQLTrack config files.

`sqltrack.config.`**find_config_file**(*path: Path | str | None = None*, *config_name='sqltrack.conf'*)

If either `path` argument or `SQLTRACK_CONFIG_PATH` environment variable is not `None`. The `path` argument takes precedence if both are defined.

Otherwise, if no explicit path is given, these default location are checked, in this order:

1. `./sqltrack.conf`
2. `$XDG_CONFIG_HOME/sqltrack/sqltrack.conf`
3. `%APPDATA%/sqltrack/sqltrack.conf`
4. `$HOME/.config/sqltrack/sqltrack.conf`
5. `$HOME/sqltrack.conf`

The first path that exists is returned, or `None` if none can be found.

`sqltrack.config.`**get_env_config**() → dict

Load config from environment variables. Variables names need to match `SQLTRACK_DSN_<PARAM>`, where `<PARAM>` is converted to lowercase in the returned dictionary.

`sqltrack.config.`**load_config**(*path: Path | str | None = None*, *config_name='sqltrack.conf'*, ***kwargs*) → dict

Locates and parses config files, as well as other sources which can add or override parameters.

find_config_file() with the given `path` argument is used to determine which config file to load. `ValueError` is raised if an explicitly specified config file (by either the `path` argument or the `SQLTRACK_CONFIG_PATH` environment variable) does not exist.

The final config is assembled as follows, starting with the the default config:

```
DEFAULT_CONFIG = {
    "engine": "postgres",
    "dbpath": "sqltrack.db",
    "user": getpass.getuser(),
    "dbname": getpass.getuser(),
}
```

Parameters are added or overridden in this order:

1. Parsed from config file, if any.
2. From `SQLTRACK_DSN_<PARAM>` environment variables. (see *get_env_config()* for details).
3. `kwargs` given to this function.

3.8.5 sqltrack.engines package

`sqltrack.engines.`**create_engine**(*config: dict*) → *Engine*

Subpackages

sqltrack.engines.postgres package

This module provides functionality to enable using PostgreSQL as backend.

class `sqltrack.engines.postgres.`**PostgresEngine**(*config*)

Bases: *Engine*

connect(*cursor_factory=<class 'psycopg.Cursor'>*)

Returns database connection objects.

data_dir()

Returns the path to the directory where supporting files for this engine are stored.

map_type(*client*, *typ*)

Given Python type `typ`, returns SQL type as string.

sqltrack.engines.sqlite package

This module provides functionality to enable using SQLite as backend.

class `sqltrack.engines.sqlite.`**SQLiteEngine**(*config*)

Bases: *Engine*

connect()

Returns database connection objects.

data_dir()

Returns the path to the directory where supporting files for this engine are stored.

map_type(*client*, *typ*)

Given Python type `typ`, returns SQL type as string.

Submodules

sqltrack.engines.engine module

class sqltrack.engines.engine.**Engine**

Bases: ABC

abstract connect() → DBAPIConnection

Returns database connection objects.

abstract data_dir() → str

Returns the path to the directory where supporting files for this engine are stored.

abstract map_type(*client*, *typ*) → str

Given Python type `typ`, returns SQL type as string.

schema = None

sqltrack.engines.json module

sqltrack.engines.json.**jsonb**(*obj*)

Return `Jsonb(obj)()` if it is not None.

3.8.6 sqltrack.notebook module

sqltrack.notebook.**format_dataframe**(*df: DataFrame*, *formatting: dict | None = None*, *na_rep='--'*, *relative_datetimes: bool = True*, *string_ellipsis: str | bool = 'left'*) → str

Returns a copy of the given Pandas DataFrame with formatting applied.

By default the following functions are applied to these the following columns (case-insensitive):

- `" "`: *format_marked()*
- `"m"`: *format_marked()*
- `"marked"`: *format_marked()*
- `"s"`: *format_status()*
- `"status"`: *format_status()*
- `"tags"`: *format_tags()*
- `"progress"`: *format_percentage()*

If the column name is not found in the formatting function dictionary, then the formatting function is selected based based on dtype:

- Any datetime-like: *format_datetime_relative()* or *format_datetime()* if relative_datetimes is False
- Any str-like: format_string() with the given string_ellipsis parameter

Parameters

- **formatting** – overwrite the default format functions for named columns; names are case-insensitive
- **relative_datetimes** – if True (default), use *format_datetime_relative()* for columns with datetime-like dtype, else *format_datetime()*
- **string_ellipsis** – passed as ellipsis parameter to format_string() for columns with str-like dtype

sqltrack.notebook.**format_datetime**(*dt: datetime*, *sep=' '*, *timespec='seconds'*, *na_rep='--'*) → str

Return datetime in ISO format.

Parameters

- **dt** – datetime to format
- **sep** – date and time separator
- **timespec** – precision of the time part, one of 'auto', 'hours', 'minutes', 'seconds', 'milliseconds' and 'microseconds'
- **na_rep** – replacement string for NaN values

sqltrack.notebook.**format_datetime_relative**(*dt: datetime*, *sep=' '*, *timespec='seconds'*, *na_rep='--'*) → str

Return time since given datetime in human-readable form.

Parameters

- **dt** – datetime to format
- **sep** – date and time separator
- **timespec** – precision of the time part, one of 'auto', 'hours', 'minutes', 'seconds', 'milliseconds' and 'microseconds'
- **na_rep** – replacement string for NaN values

sqltrack.notebook.**format_float**(*v: float*, *spec='.2f'*, *na_rep='--'*) → str

Format a float value.

Parameters

- **v** – float value to format
- **spec** – format spec; default ".2f"
- **na_rep** – replacement string for NaN values

sqltrack.notebook.**format_marked**(*is_marked: bool*)

If is_marked is True, return a gold star, else empty string.

sqltrack.notebook.**format_percentage**(*v*, *mul=100*, *spec='.1f'*, *na_rep='--'*) → str

Returns a percentage value with bar in background.

Parameters

- **v** – percentage value to format

- **mul** – multiplicative factor for display; defaults to 100 for float values in [0,1]
- **spec** – format spec; default ".1f"
- **na_rep** – replacement string for NaN values

sqltrack.notebook.**format_status**(*status: str*)

Return an icon for the given status.

sqltrack.notebook.**format_tags**(*tags: dict*)

Return tag bubbles for the given tags.

sqltrack.notebook.**format_timedelta**(*td: timedelta*, *na_rep='--'*) → str

Return a timedelta in human-readable form.

Parameters

- **td** – timedelta to format
- **na_rep** – replacement string for NaN values

sqltrack.notebook.**init_notebook_mode**()

Add the sqltrack stylesheet to the notebook.

sqltrack.notebook.**textcolor**(*text: str*, *colors: Sequence[str] = None*)

Return a color name for the given text, based on its hash value.

3.8.7 sqltrack.pandas module

sqltrack.pandas.**query_dataframe**(*client: Client*, *query: str | SQL*, *parameters=()*) → DataFrame

Run a query and return the result as a Pandas DataFrame.

Parameters

- **client** – The DB client to use
- **query** – The query to retrieve data
- **parameters** – Optional set of parameters passed to the cursor

3.8.8 sqltrack.queries module

sqltrack.queries.**first_row**(*client: Client*, *query: str | SQL*, *parameters=()*)

Execute a query and return the first matching row, if any.

Parameters

- **cursor** – Cursor to use
- **query** – Query to execute
- **parameters** – Optional parameters

sqltrack.queries.**first_value**(*client: Client*, *query: str | SQL*, *parameters=()*)

Execute a query and return the first value of the first matching row, if any.

Parameters

- **cursor** – Cursor to use
- **query** – Query to execute

- **parameters** – Optional parameters

sqltrack.queries.**first_values**(*client:* Client, *query: str* | SQL, *parameters=()*)

Execute a query and return the first value of each matching row.

Parameters

- **cursor** – Cursor to use
- **query** – Query to execute
- **parameters** – Optional parameters

3.8.9 sqltrack.sigterm module

sqltrack.sigterm.**deregister**(*key*)

sqltrack.sigterm.**register**(*key*, *func*)

3.8.10 sqltrack.util module

class sqltrack.util.**SQL**(*obj: LiteralString*)

Bases: Composable

A *Composable* representing a snippet of SQL statement.

!SQL exposes *join()* and *format()* methods useful to create a template where to merge variable parts of a query (for instance field or table names).

The *!obj* string doesn't undergo any form of escaping, so it is not suitable to represent variable identifiers or values: you should only use it to pass constant strings representing templates or snippets of SQL statements; use other objects such as *Identifier* or *Literal* to represent variable parts.

Example:

```
>>> query = sql.SQL("SELECT {0} FROM {1}").format(
...     sql.SQL(', ').join([sql.Identifier('foo'), sql.Identifier('bar')]),
...     sql.Identifier('table'))
>>> print(query.as_string(conn))
SELECT "foo", "bar" FROM "table"
```

as_bytes(*context: AdaptContext* | *None*) → bytes

Return the value of the object as bytes.

Parameters
context (*connection* or *cursor*) – the context to evaluate the object into.

The method is automatically invoked by *~psycopg.Cursor.execute()*, *~psycopg.Cursor.executemany()*, *~psycopg.Cursor.copy()* if a *!Composable* is passed instead of the query string.

as_string(*context: AdaptContext* | *None*) → str

Return the value of the object as string.

Parameters
context (*connection* or *cursor*) – the context to evaluate the string into.

format(**args: Any*, ***kwargs: Any*) → Composed

Merge *Composable* objects into a template.

Parameters

- **args** – parameters to replace to numbered (`{0}`, `{1}`) or auto-numbered (`{}`) placeholders
- **kwargs** – parameters to replace to named (`{name}`) placeholders

Returns
the union of the *!SQL* string with placeholders replaced

Return type
Composed

The method is similar to the Python *str.format()* method: the string template supports auto-numbered (`{}`), numbered (`{0}`, `{1}`...), and named placeholders (`{name}`), with positional arguments replacing the numbered placeholders and keywords replacing the named ones. However placeholder modifiers (`{0!r}`, `{0:<10}`) are not supported.

If a *!Composable* objects is passed to the template it will be merged according to its *as_string()* method. If any other Python object is passed, it will be wrapped in a *Literal* object and so escaped according to SQL rules.

Example:

```
>>> print(sql.SQL("SELECT * FROM {} WHERE {} = %s")
...     .format(sql.Identifier('people'), sql.Identifier('id'))
...     .as_string(conn))
SELECT * FROM "people" WHERE "id" = %s

>>> print(sql.SQL("SELECT * FROM {tbl} WHERE name = {name}")
...     .format(tbl=sql.Identifier('people'), name="O'Rourke"))
...     .as_string(conn))
SELECT * FROM "people" WHERE name = 'O''Rourke'
```

join(*seq: Iterable[Composable]*) → Composed

Join a sequence of *Composable*.

Parameters
seq (iterable of *!Composable*) – the elements to join.

Use the *!SQL* object's string to separate the elements in *!seq*. Note that *Composed* objects are iterable too, so they can be used as argument for this method.

Example:

```
>>> snip = sql.SQL(', ').join(
...     sql.Identifier(n) for n in ['foo', 'bar', 'baz'])
>>> print(snip.as_string(conn))
"foo", "bar", "baz"
```

sqltrack.util.**coalesce**(**values*) → object

Returns the first none-None value.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

L

M

P

Q

R

S

T