SQLTrack

Joachim Folz

Mar 15, 2023

CONTENTS

SQLTrack is a set of tools to track your (machine learning) experiments. While using other tools like Sacred or mlflow tracking, we found that they limited how we could track our experiments and later what analyses we could perform. What we realized is that it is ultimately futile for library authors to guess how experiment data will be used. If it was possible for a library to cater to every single use case it would then become too bloated to use.

That is why our goal is to collect a wide variety of examples for analyses and visualizations to empower our users, instead of providing complex functionality in our package.

SQLTrack provides a basic schema for experiments, runs, and metrics that you can extend to suit your needs, along with some basic tools to set up the database and store experiment data.

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

```
from random import random

import sqltrack
from sqltrack.commands import setup

def main():
    client = sqltrack.Client()
    setup(client, [(
        "test",
        "ALTER TABLE metrics ADD COLUMN IF NOT EXISTS loss FLOAT;"
    )])
    experiment = sqltrack.Experiment(client, name="Random")
    run = experiment.get_run()
    with run.track():
        for epoch in range(90):
            run.add_metrics(step=epoch, loss=random())

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

CHAPTER

ONE

CONTENTS

1.1 Getting started

Currently SQLTrack supports PostgreSQL through the psycopg driver. We don't plan on adding support forany other databases, except SQLite if there is demand for it. We've tried using ORMs, but found that they made things way more complicated than they needed to be and - most importantly - they obfuscated the DB schema from us. 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.

1.1.1 Installation

SQLTrack can be installed like any other Python package, e.g., `pip install sqltrack`. By default only core dependencies are installed, which speeds up usage in containerized environments. Core functionality located in the toplevel *sqltrack* package allows tracking experiments and working with the database. To use some of the convenience functions for anaylsis later, install the full package with `pip install sqltrack[full]`.

On Linux, your distribution repositories should include a version of PostgreSQL you can use. There are also install instructions for MacOS and Windows. We develop against PostgreSQL 13, but any currently supported version should work.

1.1.2 Database creation

Todo: Creating a database:

```
CREATE DATABASE ${USERNAME};
```

Create a user:

```
CREATE USER ${USERNAME};
REVOKE CONNECT ON DATABASE ${USERNAME} FROM PUBLIC;
GRANT ALL PRIVILEGES ON DATABASE ${USERNAME} to ${USERNAME} WITH GRANT OPTION;
```

Authentication

Create a schema:

```
CREATE SCHEMA IF NOT EXISTS ${SCHEMA};
```

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};
```

1.1.3 Base schema

This is the basic schema SQLTrack defines (minus some details like indexes), with tables `experiments`, `experiment_links`, `runs`, `run_links`, and `metrics`.

`runs.status` has the custom enum type `runstatus`. It behaves like text when used with the psycopg driver. Possible values have been lifted from Slurm job status.

```sql
BEGIN;

CREATE TABLE applied_migrations (
    name TEXT,
    PRIMARY KEY(name)
);

CREATE TABLE experiments (
        id BIGINT GENERATED BY DEFAULT AS IDENTITY,
        time_created TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP,
        name TEXT NOT NULL,
    comment TEXT,
    tags JSONB,
        PRIMARY KEY (id),
    UNIQUE(name)
);

CREATE INDEX experiments_tags_gin ON experiments USING GIN (tags);
CREATE INDEX experiments_tags_gin_path ON experiments USING GIN (tags jsonb_path_ops);

CREATE TABLE experiment_links (
        from_id BIGINT NOT NULL,
        kind TEXT NOT NULL,
    to_id BIGINT NOT NULL,
        PRIMARY KEY(from_id, kind, to_id),
        FOREIGN KEY(from_id) REFERENCES experiments(id),
        FOREIGN KEY(to_id) REFERENCES experiments(id)
);

CREATE TYPE runstatus AS ENUM (
    'BOOT_FAIL',
    'CANCELLED',
    'CONFIGURING',
    'COMPLETED',
    'COMPLETING',
    'DEADLINE',
    'FAILED',
    'NODE_FAIL',
    'OUT_OF_MEMORY',
```

(continues on next page)

(continued from previous page)

```
    'PENDING',
    'PREEMPTED',
    'RESV_DEL_HOLD',
    'REQUEUE_FED',
    'REQUEUE_HOLD',
    'REQUEUED',
    'RESIZING',
    'REVOKED',
    'RUNNING',
    'SIGNALING',
    'SPECIAL_EXIT',
    'STAGE_OUT',
    'STOPPED',
    'SUSPENDED',
    'TIMEOUT'
);

CREATE TABLE runs (
        id BIGINT GENERATED BY DEFAULT AS IDENTITY,
        experiment_id BIGINT NOT NULL,
        status runstatus NOT NULL DEFAULT 'PENDING',
        time_created TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP,
        time_started TIMESTAMP WITH TIME ZONE,
        time_updated TIMESTAMP WITH TIME ZONE,
    comment TEXT,
    tags JSONB,
    args JSONB,
    env JSONB,
        PRIMARY KEY(id),
        FOREIGN KEY(experiment_id) REFERENCES experiments(id) ON DELETE CASCADE
);

CREATE INDEX runs_tags_gin ON runs USING GIN (tags);
CREATE INDEX runs_tags_gin_path ON runs USING GIN (tags jsonb_path_ops);
CREATE INDEX runs_args_gin ON runs USING GIN (args);
CREATE INDEX runs_args_gin_path ON runs USING GIN (args jsonb_path_ops);
CREATE INDEX runs_env_gin ON runs USING GIN (env);
CREATE INDEX runs_env_gin_path ON runs USING GIN (env jsonb_path_ops);

CREATE TABLE run_links (
        from_id BIGINT NOT NULL,
        kind TEXT NOT NULL,
    to_id BIGINT NOT NULL,
        PRIMARY KEY(from_id, kind, to_id),
        FOREIGN KEY(from_id) REFERENCES runs(id) ON DELETE CASCADE,
        FOREIGN KEY(to_id) REFERENCES runs(id) ON DELETE CASCADE
);

CREATE TABLE metrics (
        run_id INTEGER NOT NULL,
        step BIGINT NOT NULL DEFAULT 0,
        progress DOUBLE PRECISION NULL DEFAULT 0.0,
```

(continues on next page)

(continued from previous page)

```
    PRIMARY KEY (run_id, step, progress),
        FOREIGN KEY(run_id) REFERENCES runs(id) ON DELETE CASCADE
);

END;
```

1.1.4 Defining metrics

Note that the `metrics` table doesn't contain any columns to store metrics yet. Users need to add these as required. E.g., a script to add 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,
        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;
```

Now you might ask why we make you add columns for your metrics, because that might seem 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. Also, 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.

Put your instructions to add metrics columns etc. in a SQL script file, e.g. `v001.sql`, for use later. Add `v002.sql` etc. to update your schema.

1.1.5 Setup the database

SQLTrack provides a simple tool to setup your database.

```
$ sqltrack -h
usage: sqltrack [-h] [-u USER] [-a HOST] [-d DATABASE] [-s SCHEMA]
                [-c CONFIG_PATH]
                {setup} ...

positional arguments:
  {setup}               Available commands.
    setup               Setup (and update) the database.

options:
  -h, --help            show this help message and exit
  -u USER, --user USER  username
  -a HOST, --host HOST  DB host (and port)
  -d DATABASE, --database DATABASE
                        database name
  -s SCHEMA, --schema SCHEMA
                        schema name
  -c CONFIG_PATH, --config-path CONFIG_PATH
                        path to config file
```

User, host, database, 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>
database=<DATABASE>
schema=<SCHEMA>
```

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.

1.1.6 Track an experiment

```python
from random import random
import sqltrack

def main():
    client = sqltrack.Client()
    experiment = sqltrack.Experiment(client, name="Very science, much data")
    run = experiment.get_run()
    with run.track():
        for epoch in range(90):
            metrics = {"train_loss": random(), "train_top1": random()}
            run.add_metrics(step=epoch, progress=epoch/epochs, **metrics)
```

1.1.7 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 <https://dash.plotly.com/introduction>* 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*.

1.2 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.

1.2.1 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 in 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]:
```python
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 Postgres 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, this time of run state, progress, and tags. Additionally, we also tell it to format `val_top1` as a percentage.

[3]:
```python
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,
```

(continues on next page)

(continued from previous page)

```
            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,
```

(continues on next page)

(continued from previous page)

```
                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,
                        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(
```

(continues on next page)

(continued from previous page)

```
        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
```

1.2.2 Plotly Dash

Todo: Create a custom UI with Dash.

1.3 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 queued 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...]

    options:
        -h --help               Print help text.
        --model M               Which model to train [default: resnet18]
```

(continues on next page)

(continued from previous page)

```
        -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.batch_size)

# 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'}
None
{'amp': True,
 'batch_size': None,
 'epochs': 360,
 'help': False,
 'learning_rate': ['0.1'],
 'model': 'resnet18'}
None
```

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 is 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.

1.3.1 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.

Tip: 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.

1.3.2 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]]`.

1.4 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**

1.5 API reference

1.5.1 sqltrack package

Module contents

class sqltrack.**Client**(*config_path: str = None*, ***kwargs*)

Bases: `object`

Creates and manages `psycopg.Connection` objects when used as a context manager:

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

Alternatively, if you don't need to use the connection directly, you can also get a cursor:

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

Connection parameters are given as `kwargs`. Common options are `user`, `dbname`, `host`, and `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`.

Experiment and Run objects obtain connections as required. Nested contexts reuse the same connection (reentrant), so they can be used to avoid connecting to the database multiple times over a short period. E.g., the following snippet will connect only once, with the caveat that everything happens within the same transaction:

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

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

Parameters

- **config_path** – 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.

connect() → Connection

Context manager that connects to the DB. Use in with statement:

```
with client.connect() as conn:
    ... connection things ...
    with client.cursor() as cursor:
```

(continues on next page)

(continued from previous page)

```
        ... cursor things ...
    ... connection things ...
```

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

Nested contexts reuse the same connection (reentrant), so they can be used to avoid connecting to the database multiple times over a short period. E.g., the following snippet will connect only once, with the caveat that everything happens within the same transaction:

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

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

cursor() → Cursor

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.

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*, *comment: str | None = None*, *tags: Iterable[str] | None = None*)

Bases: `object`

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

Note: Parameters comment and tags are ignored if the experiment already exists.

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
- **comment** – text comment; See `experiment_set_comment()` for details
- **tags** – list of experiment tags; See `experiment_set_tags()` for details

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

Add a link to another experiment.

add_tags(**tags: str*)

Add tags to the experiment.

get_run(*run_id: int | str | None = None, status: str | None = None, comment: str | None = None, tags: Iterable[str] | None = None, args: 'auto' | dict | None = None, env: 'auto' | dict | None = None, updated: 'auto' | datetime | None = None*) → *Run*

Get a Run object, see its documentation for more details.

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

Remove a link to another experiment.

remove_tags(**tags: str*)

Remove tags from the experiment.

set_comment(*comment: str*)

Set the experiment comment.

set_name(*name: str*)

Set the experiment name.

set_tags(**tags: str*)

Set tags of the experiment.

class sqltrack.**Run**(*client: Client, experiment_id: int | None = None, run_id: int | str | None = None, status: str | None = None, comment: str | None = None, tags: Iterable[str] | None = None, args: 'auto' | dict | None = None, env: 'auto' | dict | None = None, updated: 'auto' | datetime | None = None*)

Bases: `object`

Helper class to manage runs.

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

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
- **comment** – just some text comment; See `run_set_comment()` for details
- **tags** – list of run tags; See `run_set_tags()` for details
- **args** – run parameters; See `run_set_args()` for details
- **env** – environment variables; See `run_set_env()` for details
- **updated** – update timestamp; See `run_set_updated()` for details

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

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

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

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

add_link(*kind: str*, *to_id: int*)
Add a link to another run.

add_metrics(*step: int = 0*, *progress: float = 0.0*, *set_updated: 'auto' | datetime | None = 'auto'*, ***metrics*)
Add metrics to the run. See `run_add_metrics()` for details.

add_tags(**tags: str*)
Add tags to the run.

remove_args(*args: 'auto' | dict | None = 'auto'*)
Remove parameters from the run. See `run_remove_args()` for details.

remove_env(*env: 'auto' | dict | None = 'auto'*)
Remove environment variables from the run. See `run_remove_env()` for details.

remove_link(*kind: str*, *to_id: int*)
Remove a link to another run.

remove_tags(**tags: str*)
Remove tags from the run.

set_args(*args: 'auto' | dict | None = 'auto'*)
Set the run parameters. See `run_set_args()` for details.

set_comment(*comment: str*)
Set the run comment.

set_created(*dt: 'auto' | datetime | None = 'auto'*)
Set time_created for the run. See `run_set_created()` for details.

set_env(*env: 'auto' | dict | None = 'auto'*)
Set the run environment. See `run_set_env()` for details.

set_started(*dt: 'auto' | datetime | None = 'auto'*)
Set time_started for the run. See `run_set_started()` for details.

set_status(*status: str*)
Set the run status. See `run_set_status()` for details.

set_tags(**tags: str*)
Set the run tags.

set_updated(*dt: 'auto' | datetime | None = 'auto'*)
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'*, *args: 'auto' | dict | None = None*, *env: 'auto' | dict | None = None*)
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
- **env** – control what to do about the run's env; See `run_set_env()` for details

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

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'*, *args: 'auto' | dict | None = None*, *env: 'auto' | dict | None = None*)

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
- **`env`** – control what to do about the run's env; See *`run_set_env()`* 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*)

Set an experiment comment.

`sqltrack.`**`experiment_set_name`**(*client: Client*, *experiment_id: int*, *name: str | None*)

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 parameters 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 parameters, may be `'auto'` (detect parameters), `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 (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*, *step: int = 0*, *progress: float = 0.0*, ***metrics*)

Add metrics to a run.

`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 parameters from an existing run.

Parameters

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

`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_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 parameters. 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 parameters, may be `'auto'` (detect parameters), `dict`, or None (do nothing)

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

Set the comment for an existing run. If comment is None, do nothing.

`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 (do nothing)

`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 | None)`

Set a run status to one of the following:

- `BOOT_FAIL`
- `CANCELLED`
- `CONFIGURING`
- `COMPLETED`
- `COMPLETING`
- `DEADLINE`

- `FAILED`
- `NODE_FAIL`
- `OUT_OF_MEMORY`
- `PENDING`
- `PREEMPTED`
- `RESV_DEL_HOLD`
- `REQUEUE_FED`
- `REQUEUE_HOLD`
- `REQUEUED`
- `RESIZING`
- `REVOKED`
- `RUNNING`
- `SIGNALING`
- `SPECIAL_EXIT`
- `STAGE_OUT`
- `STOPPED`
- `SUSPENDED`
- `TIMEOUT`

Does nothing if status is None.

`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)

1.5.2 sqltrack.args module

sqltrack.args.**convert_argument**(*name: str*, *value: str*) → object

If the given value is type `str()`, try to apply all registered argument conversions in order. If no conversion works the original string is returned.

By default conversions are attempted as follows:

- `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()`

You can register new conversions (or replace existing ones) with `register_conversion()`. The final trial and error stage is fixed.

sqltrack.args.**convert_arguments**(*args: dict*, *simplify_names=True*) → ParsedOptions | None

Apply `convert_argument()` to all argument values in the given dictionary. Returns a new `docopt.ParsedOptions` dictionary with converted arguments, or None if input was None.

If `simplify_names` is True argument names are simplified. Leading dashes are stripped, any remaining dashes are replaced with underscores, and angle brackets are removed.

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

1.5.3 sqltrack.commands package

Submodules

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 filename, 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

1.5.4 sqltrack.notebook module

`sqltrack.notebook.format_bool`(*b: bool*, *na_rep='--'*) → str

Return bool value.

Parameters

- **b** – bool to format
- **na_rep** – replacement string for NaN values

`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_string`(*s: str*, *ellipsis='left'*, *na_rep='--'*) → str

Return string wrapped to display long text with ellipsis.

Parameters

- **s** – string to wrap
- **ellipsis** – if "left" (default), ellipsis is placed on the left and the end is displayed fully; if True, ellipsis is placed on the right and the beginning of the string is displayed; if False, the string is returne as-is
- **na_rep** – replacement string for NaN values

`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.

1.5.5 sqltrack.pandas module

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

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

Parameters

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

1.5.6 sqltrack.queries module

sqltrack.queries.**first_row**(*cursor: Cursor*, *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**(*cursor: Cursor*, *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**(*cursor: Cursor*, *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

1.5.7 sqltrack.sigterm module

1.5.8 sqltrack.util module

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

Returns the first none-None value.

sqltrack.util.**load_config**(*path=None*) → dict

Parse config file. Parameters may be overridden by `SQLTRACK_DSN_<PARAM>` environment variables. Path defaults to `./sqltrack.conf`.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

S

T