OpenRefine-Wikibase reconciliation interface

Release 1.0

Antonin Delpeuch

Nov 10, 2022

CONTENTS:

This software offers a reconciliation interface for a Wikibase instance, following the specifications of the reconciliation API.

This manual is intended for developers and Wikibase administrators. Reconciliation users should instead refer to the main page of the corresponding reconciliation endpoint, where user documentation can be found. For Wikidata, this is available at https://wikidata.reconci.link/.

CHAPTER

ONE

INSTALLING THE RECONCILIATION SERVICE

1.1 Requirements

The Wikibase instance should have:

- An associated SPARQL query service;
- Some special properties and items to represent its type system, by analogy to the one in place in Wikidata with instance of (P31) and subclass of (P279), with a root type such as entity (Q35120);

In addition it is also recommended that the Wikibase instance uses the CirrusSearch extension (ElasticSearch-based search engine).

1.2 Configuration

The configuration of the service is done in a Python file *config.py*. A sample configuration file is provided for Wikidata, *config_wikidata.py*.

1.3 Installing with Docker

You can run this service with docker. First, clone the repository and go to its root directory:

```
git clone https://github.com/wetneb/openrefine-wikibase
cd openrefine-wikibase
```

Then, copy the sample *config_docker.py* to *config.py* and modify the copy to point to the Wikibase instance of your choice.

Finally, start the service:

```
docker-compose up
```

On Windows you will need to accept the Windows Firewall popup to expose the port 8000 where the service runs.

You can then access the landing page of your new reconciliation service at *http://localhost:8000/*.

To use it in OpenRefine, you can add the reconciliation service (in the "Start reconciling" dialog) with the address "http://localhost:8000/en/api". You can then use this reconciliation service to match data to items stored in your Wikibase instance.

1.4 Installing manually

It is possible to run this web service locally. You will need Python 3.7 or later and a redis instance.

- Clone this repository, either with git (*git clone https://github.com/wetneb/openrefine-wikibase*) or by downloading the repository from Github as an archive
- It is recommended to set up a virtualenv to isolate the dependencies of the software from the other python packages installed on your computer. On a UNIX system, *python3 -m venv .venv* and *source .venv/bin/activate* will do. On a Windows system, *python.exe -m venv venvname* followed by *venvnameScriptsactivate* should work.
- Install Python3 development packages (libpython3-dev on Debian based systems)
- Install the Python dependencies with *pip install -r requirements.txt*
- Copy the configuration file provided: *cp config_wikidata.py config.py* (*copy config_wikidata.py config.py* on Windows)
- Edit the configuration file *config.py* so that *redis_client* contains the correct settings to access your redis instance. The default parameters should be fine if you are running redis locally on the default port.
- Finally, run the instance with *python app.py* (for development purposes). The service will be available at *http://localhost:8000/en/api*.

On Debian-based systems, it looks as follows:

```
sudo apt install git redis-server python3 virtualenv libpython3-dev
git clone https://github.com/wetneb/openrefine-wikibase
cd openrefine-wikibase
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

1.5 Deploying in production

To run this service in production, we recommend using *gunicorn* in conjunction with *uvicorn*. Those packages can be installed in the same virtual environment as the code, with *pip install gunicorn uvicorn*.

The web service can then be run with *gunicorn app:app -b localhost:8080 –workers 4 –worker-class uvicorn.workers.UvicornWorker*.

Since this process needs to keep running, you should deploy it appropriately, for instance in a Kubernetes pod or as a systemd service. Here is an example systemd service configuration file, stored in */etc/systemd/system/wdrecon.service*:

```
[Unit]
Description=Wikidata reconciliation service
After=network.target

[Service]
Type=simple
User=wdrecon
Group=wdrecon
Restart=always
EnvironmentFile=-/etc/default/wdrecon
WorkingDirectory=/home/wdrecon/openrefine-wikibase/
ExecStart=/bin/sh -c '${WDRECON_GUNICORN_BIN} app:app -b localhost:8080 --workers $
```

(continues on next page)

(continued from previous page)

```
↪{WDRECON_WORKERS} --worker-class uvicorn.workers.UvicornWorker'

[Install]
WantedBy=multi-user.target
```

This is accompanied by the following environment file, stored at */etc/default/wdrecon*:

```
WDRECON_GUNICORN_BIN="/home/wdrecon/venv/bin/gunicorn"
WDRECON_WORKERS="4"
```

For the Wikidata service, we run multiple instances of such a gunicorn server, gathered together behind an Apache load balancer.

1.6 Tips about Redis configuration

If you are in a position to configure the Redis instance you are using, then you can do the following:

- Disable snapshots of the Redis instance to disk, because this software only uses Redis as a cache which can be completely lost. This can be done by commenting out all the *save* lines in *redis.conf*;
- Set a maximum memory limit of your liking, together with an eviction policy (such as LRU), so that the redis instance does not eat up more memory than reasonable on your server. This can be done in *redis.conf* by adding directives such as *maxmemory 3gb* and *maxmemory-policy volatile-lru*.

CHAPTER

TWO

ARCHITECTURE OVERVIEW

This service acts as a thin wrapper between the reconciliation client (such as OpenRefine) and the Wikibase instance. It does not maintain a search index on its own: it relies on the existing search capabilities of the Wikibase instance (via the MediaWiki API) and the SPARQL query service.

A redis instance is used for caching data to avoid making too many queries to the Wikibase instance.

2.1 Reconciliation

Reconciliation queries are processed as follows:

- The given text (*query* field) is searched for with both search APIs provided the Wikibase instance (the auto-complete API *action=wbsearchentities*, and the search API *action=query&list=search*). For both search endpoints we only look at the first page of results. The results are merged into one list. The reason for this is that none of the two endpoints can be trusted to surface the relevantcandidates systematically. For instance, searching for "USA" in *action=wbsearchentities* will return United States of America (Q30) as first result, but with the same query in *action=query&list=search*, this entity is not present in the first page of results. Conversely, searching for "Lovelace, Ada" in *action=query&list=search* will return Ada Lovelace (Q7259), but will not yield any results with *action=wbsearchentities*.
- The contents of each candidate item is retrieved in JSON via the *wbgetentities* API action. Furthermore, the types and any other property used for reconciliation is also fetched on the candidate items (again with *wbgetentities*);
- Candidates are filtered by type. This is done by fetching the Qids of all the subclasses of the given target type (with SPARQL) and only keeping the candidates whose type is one of these subclasses;
- The candidates are scored by comparing the values supplied in the query to the values obtained in the previous step;
- The candidates are sorted by decreasing score and returned to the user.

There are exceptions to this workflow:

- When Qids or sitelinks are supplied in the *query* field, they are directly looked up accordingly (instead of being searched for with the search APIs);
- When a unique identifier is supplied as a property, candidates are first fetched by looking for items with the supplied identifiers (via SPARQL), and text search on the query is only used as a fallback.
- When no type constraint is supplied, an implicit negative type constraint is used instead (to filter out all internal items, which are marked by subclasses of Wikimedia internal item (Q17442446).

Calls to the API are done in parallel, up to a limit of maximum concurrent queries to avoid overloading the Wikibase instance. This means that supplying queries by batch (as allowed by the protocol) can be significantly more efficient than submitting them individually.

2.2 Auto-complete (suggest) services

These services are used to provide auto-complete widgets in user interfaces around the reconciliation process. The calls to these services are directly translated to the corresponding API actions of the Wikibase instance, except for properties where the user input is also parsed as a property path beforehand (if the parsing succeeds, the parsed property path is returned as sole candidate).

2.3 Preview

Previewing entities is done by fetching data for the corresponding item and displaying a few snippets of information for the item. For Wikidata, the autodesc service is also used to generate a description automatically for the item.

2.4 Data extension

Properties requested on items are fetched in the same way as during reconciliation, by attempting to minimize the calls to the Wikibase instance (batching requested items, caching).

CHAPTER

THREE

SCORING MECHANISM

This page describes how the scores of reconciliation candidates are computed.

3.1 Stability

The scoring mechanism used in this reconciliation service can change, the specifics of its computation should not be relied on by users. Instead, we recommend that individual scoring features are used instead.

3.2 Global matching formula

The score of each candidate is obtained as a weighted sum of the scores of individual features. It ranges from 0 to 100. When no candidates can be found matching the target type, candidates of wrong or no types are also returned, with their score divided by two.

For each supplied property, all query values are matched against reference values and the maximum matching score of all pairs is used as the similarity score for this property.

3.3 Name matching

Two names (such as an item label and a query) are matched by token-based fuzzy matching.

3.4 Identifier matching

Values of properties which hold identifiers are matched to the queries using exact string equality (100 score if the strings are equal, 0 otherwise).

3.5 Geographical coordinate matching

Geographical coordinates are expected to be supplied in *lat,long* format (such as *53.3175,-4.6204*). The matching score peaks at 100 when the position is exactly the same and decreases linearly as the distance between the points increase, reaching 0 when the points are 1 km apart.

3.6 Date matching

The precision of Wikibase dates is taken into account when matching them against strings. Query dates are expected to be supplied in ISO format (YYYY-MM-DD) and will match the Wikibase date perfectly if they fall into the range described by the precision. It is also possible to supply query dates in YYYY-MM or YYYY format.

3.7 Quantity matching

Integer quantities are matched (score 100) if they are equal, and have a 0 score otherwise. For floating-point numbers, the score peaks at 100 for exact equality and follows otherwise this formula:

3.8 URL matching

URLs are canonicalized before being matched. Differences in scheme (HTTPS vs HTTP) are ignored.

CHAPTER

FOUR

TESTING INFRASTRUCTURE

The service comes with a test suite that can be invoked with:

```
pytest
```

To measure the coverage, you can run:

```
coverage run --omit=.venv -m pytest
```

CHAPTER

FIVE

DOCUMENTING

This manual is written using Sphinx and the source files can be found in the *docs* folder of the repository for this application. Any contribution to the docs are of course most welcome, as are any suggestions to improve the coverage of a particular subject.

In addition to this manual for developers, each reconciliation service instance offers user docs on its main page.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search