

Poliqarp 2.0 - Operator manual

Bartosz Zaborowski

April, 2016

Contents

1	Installation	3
1.1	Main modules	3
1.2	Console client	4
1.3	GUI	4
1.4	Converters	5
2	Configuration options, search engine setup	6
2.1	Main modules configuration	6
2.1.1	common	6
2.1.2	corpora	7
2.1.3	top-level settings	7
2.1.4	indexer	8
2.1.5	query_execution_optimizer	10
2.1.6	query_runner	11
2.1.7	poliqarp_server	11
2.1.8	session	12
2.1.9	storage_engine	13
2.1.10	users	14
2.2	Client configuration	15
2.2.1	GUI	16
2.2.2	console client	16
2.3	Local machine setup	16
2.4	Server setup	17
3	Corpus importing	18
3.1	Input file format	18
3.1.1	a feature structure	18
3.1.2	an indexable	19
3.1.3	a node	19
3.1.4	an edge	19
3.1.5	a secondary edge	19
3.1.6	a largescale node	20
3.1.7	a list	20

3.1.8	an amblist	20
3.1.9	a literal node	20
3.1.10	a literal	20
3.1.11	a pointer	21
3.2	Creating a new converter	21
3.2.1	A converter assembly	21
3.2.2	How to begin writing a new converter	21
3.3	Loading and indexing	22
3.3.1	Loading a corpus	22
3.3.2	Indexing a corpus	22

Chapter 1

Installation

Poliqarp 2.0 search engine comes in a few separate packages. A package `poliqarp-2.0.tar.gz` contains source code for the main modules - the search engine itself, a corpus importing tool (*reader*), an indexer module, and a database module (*storage*). Additional packages contain user interfaces (GUI and console client) and corpus converters from various formats to the Poliarp2 input format. The sections below describe steps needed to install particular packages.

1.1 Main modules

The main modules package contains source code that needs to be compiled. It requires the operating system to be at least POSIX compatible (a few Linux distributions were tested, MinGW and other *NIX-es may require some tweaking). The building system bases on *GNU autotools*. Thus, the installation is pretty standard, it consists of the following steps:

```
tar xzvf poliqarp-2.0.tar.gz
cd poliqarp-2.0
./configure
make
sudo make install
```

The last step may require superuser privileges. The *configure* script checks if all the requirements are met and reports if something is absent. The most important dependencies are:

- *GNU make or compatible* (build dependency)
- *binutils* (build dependency)
- C and C++ compiler (tested on GCC 4.9 and GCC 5.3) (build dependency)

- *zlib*
- *xz-libs* (liblzma)
- *sqlite3*
- *boost* libraries (at least *boost-regex* and *boost-serialization*)
- *python3* (with libpython library)
- *libhttpserver* (you can get it from <https://github.com/etr/libhttpserver>, the version 0.9.1 is required – newer versions fail due to API changes)
- *google-perftools* (at least tcmalloc library) - optional dependency

The packages marked as (*build dependency*) are required to build the tool, but not required at run-time. Hence, when copying compiled binaries between machines (with the same CPU architecture and versions of operating systems and libraries), they are not required.

Note 1. *tcmalloc* library is optional, but highly recommended. It allows for significantly lower run-time memory usage and slightly faster execution. However, some versions (e.g. the one available in Ubuntu 15.10 repository) may cause random faults. In such cases you can disable this library by passing `--without-tcmalloc` argument to the **configure** script.

1.2 Console client

The console client (*headless client*) comes in a form of portable python scripts. Hence it does not require installation - after unpacking the tool is ready to use. The only requirements are:

- *Python3* interpreter available
- *requests* python library installed

1.3 GUI

The WEB GUI client is a portable Django application. It does not require compilation, but some setup steps are required before using it. Its dependencies are (tested versions in parentheses):

- *Python3* interpreter available (3.4)
- *requests* python library installed (2.8.1)
- *Django* WEB framework installed (1.8.3)

- *django-ckeditor* library (5.0.2)
- *django-crispy-forms* library (1.5.2)
- *psycpg2* library (2.6.1)
- *postgres* database (9.4)

The installation of the postgres database goes beyond the scope of this document. Consult the postgres homepage or your system distribution documentation for manuals on postgres installation.

The python libraries needed can be easily installed using *pip3* tool. Consult the pip3 homepage or your system distribution documentation for details on installation.

The detailed setup steps are described in a README file in the GUI package.

1.4 Converters

The converters package contains portable python scripts. They need no installation, but require the *Python3* interpreter to be installed and available.

Chapter 2

Configuration options, search engine setup

The following sections describe in details the configuration options of particular modules.

2.1 Main modules configuration

A configuration of the main modules is a *JSON* file. By default, all the main modules look for it in `~/.poliqarp/poliqarp2.conf` location. The non-standard location of the config file may be enforced by using `-c <filename>` command-line argument supported by all of the tools. The config file consists of a single JSON object with a few subsections. An example of such a file is available in an **examples** subdirectory of the main package. The following subsections discuss the available settings in details.

Note 2. *Nearly all of the settings (but the users/roles list) may be omitted or set only partially. In such cases, for the omitted entries built-in defaults are used.*

2.1.1 common

This section contains options which affect multi-threaded performance and possibility to load very large document structures.

ident_pool_size

Multi-threaded performance related setting, determines how often reader threads need to synchronize corpus-global object counters.

example

```
"ident_pool_size" : 1000
```

pools_per_largescale_node

Corpora-specific setting: `ident_pool_size*pools_per_largescale_node` must be higher than a maximal number of structures belonging to a single *largescale node* ¹. If in doubt, run the `poliqarp_reader` in test mode (with `-t` argument). If the setting is too low, the reader will fail with an informative error. If it is too high, on large corpora the reader may fail with an error related to exhausting of identifiers.

example

```
"pools_per_largescale_node" : 100
```

Note 3. Warning: *the product of the above numbers must not change after reading a corpus and is global to all corpora used in a single `poliqarp_server` instance. If changed after reading a corpus, the `poliqarp_server` will refuse to start with an appropriate error.*

2.1.2 corpora

Server document-caching related settings: debugging messages and the per-corpus cache size (in nodes). Higher `document_cache_size` value may effect in significant performance increase at a cost of significantly higher memory usage.

example

```
"debug_document_cache" : false,  
"document_cache_size" : 50000
```

2.1.3 top-level settings

debug_config

A switch to print the loaded configuration at the start.

example

```
"debug_config" : false,
```

default_locale

A global, default locale for regexes and string comparisons. Can be changed locally in queries using flags².

¹See user manual for details on available data structure types

²See user manual for details

example

```
"default_locale" : "C",
```

2.1.4 indexer

Reader and indexer related settings

batch_size_limit

Indexes for corpora are divided into batches, this value sets a size of single batch in nodes. Higher values cause longer response time (long time between starting query execution and the first results) and higher memory usage, but give overall performance increase (shorter total query execution time).

example

```
"batch_size_limit" : 1000000,
```

batches_count_per_block

For performance reasons, indexes for really large corpora may be split into separate files. Large values give similar behavior to large values of the `batch_size_limit`.

example

```
"batches_count_per_block" : 50,
```

db_flushes_per_batch

How often the reader cache is flushed – smaller values give performance boost in cost of memory.

example

```
"db_flushes_per_batch" : 50,
```

debug

Be very verbose – mainly for tracking errors in the Poliqarp2 itself.

example

```
"debug" : false,
```

index_postprocessing_buffer

An another kind of cache, this time for the indexer.

```
example
"index_postprocessing_buffer" : 10000,
```

reader_timing

Print raw clock time before progress information lines in reader.

```
example
"reader_timing" : false,
```

simple_query_concatenation_limit

index for simple queries³ may grow very fast on long non-space-delimited strings of tokens. This setting limits such cases. However, the limited index will prevent for finding such strange sentences. Usually, the values around 10 – 20 give ability to search for words like *chciałżeby* segmented as *chciał|że|by|m* in all normal linguistic contexts (e.g. *chciałżeby...*) while not bloating the index.

```
example
"simple_query_concatenation_limit" : 20,
```

summary

Show some statistics at exit of reader.

```
example
"summary" : true,
```

threads

How many threads to use when reading/indexing.

```
example
"threads" : 4,
```

³See user manual for details

verbose

Show basic info about reading/indexing progress.

example

```
"verbose" : true
```

2.1.5 query_execution_optimizer

Settings related to index-based query execution optimizer (usually the first phase of query execution).

enable_candidates_for_runner

Pass some information from optimizer to query runner to further increase performance. In case of bugs in optimizer this may affect correctness of results.

example

```
"enable_candidates_for_runner" : true,
```

lsnodemap_cache_size

Cache size for some common structures (given in batches), Large values significantly speed up an execution on large corpora, but at a cost of high memory usage.

example

```
"lsnodemap_cache_size" : 100,
```

suppress_errors

Don't report query execution optimizer messages at the console.

example

```
"suppress_errors" : false
```

other optimizer-related settings

An experimental lazy optimization drops computationally expensive parts of query which don't seem to significantly narrow down the number of results. Disabled by default – evaluations show benefits only in some cases (and slowdowns in other).

example

```
"enable_lazy_optimization" : false,  
"acceptable_laziness_factor" : 5,  
"zealousness_factor" : 10,
```

For debugging Poliqarp2: be very verbose at various stages of optimizer execution.

example

```
"debug" : false,  
"expensive_debug_counters" : false,  
"index_debug" : false,  
"print_query" : false,  
"print_time" : false,
```

2.1.6 query_runner

Settings related to query executor/runner.

max_mem_usage

Approximate per-thread memory limit (in bytes).

example

```
"max_mem_usage" : 268435456,
```

print_max_mem_usage

Be verbose.

example

```
"print_max_mem_usage" : false
```

2.1.7 poliqarp_server

poliqarp_server settings.

active_sessions_limit

How many sessions may work at the same time (globally).

example

```
"active_sessions_limit" : 3,
```

debug

Be very verbose.

example

```
"debug" : true,
```

disrupt_results_order

A solution to legal issues – deterministically disrupts results order, so there is no possibility to obtain full documents just by iterating over all results and concatenating output. Makes sense only, if the input files don't contain whole documents (otherwise a query `[] within [doc]` must return a whole document because of semantics of the query language).

example

```
"disrupt_results_order" : true,
```

listener_threads

REST API listener threads number.

example

```
"listener_threads" : 2,
```

port

REST API listening port.

example

```
"port" : 5000,
```

other server-related settings

For debugging REST API clients/connections and the Poliqarp2 itself: be verbose at various cases.

example

```
"print_requests" : false,  
"print_requests_content" : false,  
"print_worker_load" : false
```

2.1.8 session

Usage session related settings.

debug_gc

Print info on old session removal procedure.

example

```
"debug_gc" : false,
```

inactive_timeout

Session (and last query results) may be removed after that inactivity time. High numbers may cause behavior that looks like a memory leaks. Low numbers may annoy users (after that inactivity time requesting for next results will fail).

example

```
"inactive_timeout" : 1800,
```

worker_threads

How many threads to use when executing **single query**. Does not count a separate query execution optimizer thread.

example

```
"worker_threads" : 4
```

2.1.9 storage_engine

Settings related to the `poliqarp_storage`.

auth_key

Kind of a password to limit access to the database.

example

```
"auth_key" : "secret_authentication_code",
```

command_execution_timeout

Single database command timeout.

example

```
"command_execution_timeout" : 1800,
```

data_dir

Storage data directory, will be created if does not exist.

example

```
"data_dir" : "~/poliqarp/corpus_data",
```

debug

For debugging of Poliqarp2: be very verbose.

example

```
"debug" : false,
```

worker_inactive_timeout

Database worker/connection inactivity timeout (forces automatic reconnection). Low values may decrease performance, high can possibly exhaust system resources on very high load (per-process file descriptors limit in particular).

example

```
"worker_inactive_timeout" : 600
```

interface settings

Listen for connections on this interface and port.

example

```
"engine_address" : "localhost",  
"port" : 5001,
```

2.1.10 users

Settings related to REST API authorization and user/role limits. This entry in config file is a list of JSON objects. All entries in the per-user objects are required.

active_sessions_limit

This user may use only this number of sessions simultaneously.

example

```
"active_sessions_limit" : 2,
```

can_access_restricted

A list of corpora identifiers in which this user can access documents marked as *restricted access*.

```
example
"can_access_restricted" : [ "restricted_corpus_example" ],
```

login

User/role name. Used in *http basic auth* on connection of client to REST API.

```
example
"login" : "defaultuser",
```

password

User/role password. Used in *http basic auth* on connection of client to REST API.

```
example
"password" : "defaultpassword",
```

user limits

Absolute hard limits for the particular user for a single query.

```
example
"query_execution_time_limit" : 600,
"results_number_limit" : 100000
```

2.2 Client configuration

All clients must have a connection details set up correctly to work. On most local installations this does not need to be changed, however in server deployment it may require some attention. In both, the GUI and the console client, this settings may be changed by editing a `pq_client.py`. In the top of this file there are three constants: `ADDR`, `USER`, `PASSWORD`. They correspond to an URL of the server instance (IP/domain of the machine and `"server"."port"` configuration setting), a default role name and a password for this role.

2.2.1 GUI

GUI has some Django-standard configuration settings which may be changed in a `poliarp_gui/settings.py` file. Additionally, GUI offers a simple Content Management System accessible from `/admin`, which allow to set up things related to user accounts, privileges and edit additional contents of WEB GUI/start page. Details on this features can be found in the GUI manual.

2.2.2 console client

The console client has nearly no settings exposed for the user. The only interesting setting which may be altered is a `REPORT_INTERVAL` constant at the top of the `headless_pq_client.py` file. It sets an interval between progress reports to the given amount of seconds.

2.3 Local machine setup

The simplest deployment of Poliarp2 for the local usage consists of a running storage module, running server module and a running GUI interface. Assuming loaded and indexed corpora files are present in the configured `data_dir`, the starting procedure is as follows:

```
poliarp_storage -d -l storage_log_file.log
poliarp_server -d -l search_engine_log_file.log
cd GUI
python3 ./manage.py runserver
```

The above steps require the installation process from the chapter 1 to be completed. A short pause may be needed between execution of the subsequent commands.

This procedure will cause to start the database in background, then a server/search engine also in background and finally, start the web GUI on a Django-built-in tiny web-server. After a few seconds, the GUI should be available at a `http://localhost:8000/` address. The `poliarp_server` start-up may take a while if the corpora are large. Until it fully starts, the GUI may report a 503 error about inaccessible server (the page needs to be refreshed manually in such a case).

Note 4. *Since the Poliarp modules don't use any kind of encryption of connections, it is strongly recommended to use a firewall and deny all used ports from ingoing connections (by default ports 5000, 5001 and 8000).*

Note 5. *Any corpus changes (including running indexer and reader tools) require `poliarp_server` to be not running. You can terminate it either*

from some kind of system process manager or by issuing `killall poliqarp_server` command. Manual changes to the contents of `data_dir` are prohibited when the `poliqarp_storage` is running. You can terminate it similarly to terminating server: from process manager or by `killall poliqarp_storage`. The only exception from the above rules are advanced use cases with separate tool instances running on different configurations (ports and `data_dirs`).

Note 6. Any changes to configuration files won't affect running instances of tools. The tools need to be restarted for changes to apply.

Note 7. On some corpora it may be needed to increase a per-process system stack size limit for server to operate properly. Such cases should be clearly marked by people who loaded and indexed that corpora.

Note 8. When running the search engine on a very large corpora or a very large number of corpora, the `poliqarp_storage` may need a per-process open file number limit increased. The need for such setting will appear if the `poliqarp_storage` fails with a "Too many open files" error.

2.4 Server setup

In this section we assume a need for deployment of Poliqarp2 with either a public access through a WEB GUI or a publicly accessible REST API.

The basic steps for such a deployment are similar as for local setup. The important differences are:

- When configuring any kind of auto-starting procedures (e.g. `cron`) make sure, that there is a few second `sleep` between starting the storage and the server. Lack of or too short pause will cause server to fail to connect to database.
- If a machine hosting `poliqarp_storage` is not the same than a machine hosting `poliqarp_server`, make sure the connection between them has a high bandwidth, low lags and is either fully trusted or implemented by some encrypting transport (e.g. VPN, ssh tunneling). The connection between storage and server is not encrypted and has a weak authorization, so failing to fulfill the last requirement may be a security risk.
- For publicly accessible REST API it is advised to set up a proxy which enables HTTPS encrypted transport. The `poliqarp_server` tool does not support HTTPS protocol internally at the moment.
- For the best performance of WEB GUI, use some dedicated http server with a module for Django deployment (such as `apache2` and `mod_wsgi`). The best configuration should run a few instances of the Django app to enable concurrent loading of multiple search results.

Chapter 3

Corpus importing

This chapter describes procedure of importing a corpus in its original format into the search engine.

3.1 Input file format

The `poliqarp_reader` tool accepts only a Poliqarp2 corpus data format `.pqz`. The format is not very sophisticated. It is a document modeled by means of the Poliqarp2 data structures¹ serialized in JSON and compressed using `gzip`. The format is quite similar to the format of the search engine results.

A best way of creating correct `.pqz` files is to use a small python library distributed along with format converters in the `converters` package.

At the moment (format version 106) a `.pqz` file contains a single JSON object with the following entries:

- `objs` – a list of serialized document structures
- `doc_root_id` – an integer identifier of document root node
- `restricted` – a Boolean flag marking a "restricted access" document
- `poliqarp2_input_format` – an integer equal to 106

The list of serialized structures contains JSON objects of a following types:

3.1.1 a feature structure

A simplest attribute-value structure. Required fields:

- `_` – the JSON object type equal to `fs`

¹See user manual for details

- **id** – a numeric identifier of this object (unique at the scope of the whole document)
- **type** – the (poliarp2 data model) structure type (a string)
- **present_in** – a *pointer* to an amblist marking in which variants the structure is present
- **attrmap** – a dictionary (JSON object) containing the actual payload data. Keys are strings, values are either *literals* or *pointers* to some structures.

3.1.2 an indexable

A special, standalone kind of *feature structure*. Required fields: all present in *feature structure* with the following changes:

- **_** – the JSON object type equal to **indexable**

3.1.3 a node

Used for modeling syntax structures and separate words. Required fields: all present in *feature structure* with the following changes:

- **_** – the JSON object type equal to **node**
- **edges** – a *pointer* to list of outgoing edges
- **sec_edges** – a *pointer* to list of outgoing secondary edges

3.1.4 an edge

Used for modeling syntax graph. Required fields: all present in *feature structure* with the following changes:

- **_** – the JSON object type equal to **edge**
- **target** – a *pointer* to a target structure (usually a *node*)

3.1.5 a secondary edge

Used for modeling other graph relations. Required fields: all present in *edge* with the following changes:

- **_** – the JSON object type equal to **sec_edge**

3.1.6 a largescale node

A largescale segmentation node (e.g. for modeling a sentence, paragraph). Required fields: all that are required in a *node* with the following changes:

- `_` – the JSON object type equal to `largescale_node`

3.1.7 a list

List of structures. Required fields:

- `_` – the JSON object type equal to `list`
- `id` – a numeric identifier of this object (unique at the scope of the whole document)
- `values` – a JSON list of *literals* and/or *pointers* to some structures

3.1.8 an amblist

An ambiguity list of structures. Required fields:

- `_` – the JSON object type equal to `amblist`
- `id` – a numeric identifier of this object (unique at the scope of the whole document)
- `sel` – a *pointer* to a list of selected elements
- `all` – a *pointer* to a list of all elements. This list must superset the previous list.

3.1.9 a literal node

A structure graph wrapper for literal. Required fields:

- `_` – the JSON object type equal to `literal_node`
- `id` – a numeric identifier of this object (unique at the scope of the whole document)
- `value` – a *literal* holding the value

3.1.10 a literal

A literal value, nested in other structures. Required fields:

- `_` – the JSON object type equal to `literal`
- `enum` – optional string identifier of a scope of enums
- `value` – an actual value of the literal

3.1.11 a pointer

A pointer to a structure, nested in other structures. Required fields:

- `_` – the JSON object type equal to `ptr`
- `trg` – an identifier of a structure pointed

3.2 Creating a new converter

3.2.1 A converter assembly

The existing format converters are described in a `README` file in the `converters` package. Each converter consists of three main parts:

1. Original file(s) reading and parsing
2. Modeling the data by means of `poliarp2` data structures
3. Correctness checking and writing `.pqz` files

The reading and parsing is totally corpus-specific, so it won't be discussed here.

The modeling part is a place when all the logic takes place. It consists of building `poliarp2` data structures (each structure type has a corresponding python class in the `converter_types.py` library). That means in particular: creating objects (by standard python constructor), connecting nodes by edges (inserting `edge` objects into an `edges` field of `node` objects), inserting attribute values into the `attrmap` field of some objects and so on. The library has sane defaults, so it is no need for specifying e.g. variants when a corpus does not use them.

The last part, checking and writing files, is realized by a single call to an `output_doc` function from `reader_common.py` library.

3.2.2 How to begin writing a new converter

If there is no converter for a specific format yet, the only choice is to write a new converter. The best starting point in such a case is to use a converter for a somehow similar corpus type and alter the reading and modeling parts of that converter to handle a new format. The `converters` package contains also a self-documented toy-example of a converter which shows most of the work described above. This script may be a good base for writing a converter from scratch.

3.3 Loading and indexing

3.3.1 Loading a corpus

After converting to the *.pqz* format, the corpus has to be loaded by means of `poliqarp_reader` into the database. This tool has a few useful arguments:

- `-k` or `--corpus-config`: a simple dictionary for attribute names – can be used to remap attributes from the prohibited keywords or to create some kind of aliases. An example of such dictionary is included in the main package in the *examples* directory.
- `-g` or `--corpus-group`: sets a group of corpora. Setting this argument to the same string value in multiple corpora will cause that the list of previously used queries (in GUI) will be filtered to show only those corpora. This may be useful when loading a few corpora with the same structure/tagset and some other completely different corpora.
- `-t` or `--test-input-data`: a test mode – does not modify anything in database, but performs all parsing and checking for correctness of data.

The `poliqarp_reader` tool requires the `poliqarp_storage` to be running in background.

3.3.2 Indexing a corpus

After loading a corpus into the database it can be indexed to optimize performance of various query types. This is performed using `poliqarp_indexer` tool (it also requires storage to be running). The indexing phase is optional, but recommended – in many cases it can give a huge performance gain. There are three main indexes types:

- `s` – indexes for *Simple Queries*. They can be built only if the corpus follows some conventions. In particular:
 - all segment-nodes have *type* equal to `seg` and an *orth* attribute storing the orthographic form,
 - all segment-nodes have sub-segment node(s) as child(ren), the sub-segments should have *type* equal to `sub` and an *endword* boolean attribute set to `true` if and only if there is a whitespace after the particular sub-segment,
- `n` – node-level indexes. They can be in an extended version (with some information about neighbor nodes), which allow to speed-up queries asking for e.g. several subsequent nodes, such as a node-level regex.

The extended version may be disabled by a `-n` command line argument. This may be particularly useful for saving space and memory in corpora with very deep trees/structure graphs.

- 2 – second level indexes. Helps in optimization of total query run-time in large corpora, especially for precise queries (e.g. asking for a very non-frequent word). This will come at the cost of a lower responsibility (higher average time until the first result is found).

The second level indexes are based on the `s` and/or `n` indexes, so they should be created after those are built.