Extreme Search® Manual

Version 2024.10

https://lewis-rhodes.com

support@lewis-rhodes.com

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 ints 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': {}}
```